Universidade Federal de Santa Catarina - UFSC

Uma arquitetura para a atuação de agentes inteligentes

Henrique Prandi

Florianópolis, 2017

Universidade Federal de Santa Catarina - UFSC
Centro Tecnológico
Departamento de Informática e Estatística
Curso de Sistemas de Informação

Uma arquitetura para a atuação de agentes inteligentes

Henrique Prandi

Orientador: Prof. Dr. Elder Rizzon Santos

Trabalho de conclusão de curso apresentado à Universidade Federal de Santa Catarina para obtenção do grau em Bacharel em Sistemas de Informação

Florianópolis, 2017

Henrique Prandi

Uma arquitetura para a atuação de agentes inteligentes

Trabalho de conclusão de curso apresentado à Universidade Federal de Santa Catarina para obtenção do grau em Bacharel em Sistemas de Informação.

_____
Prof. Chefe, Cristian Koliver
Coordenador do Curso

Banca examinadora:

_____
Prof. Dr. Elder Rizzon Santos
Orientador

_____
Prof. Dr. Ricardo Azambuja Silveira

_____
Me. Thiago Ângelo Gelaim

Florianópolis, 2017

AGRADECIMENTOS

Dedico este trabalho primeiramente a meus amados pais Arli Prandi e Roseneide Prandi, que sempre foram meu exemplo de dedicação, minha fonte de sabedoria e sustentáculos fundamentais em minha caminhada durante a graduação e também ao meu irmão Guilherme Prandi por todo o apoio durante esta fase.

Também dedicado a todos os meus amigos que forneceram a base que precisei, especialmente para Marina Coelho com quem aprendi que a melhor pessoa que podemos ser é exatamente quem nós somos, e também a Daniel Yoshizawa com quem aprendi a ser paciente e aceitar aquilo que não podemos mudar. Vocês foram essenciais nesta jornada.

"Niguém baterá tão forte quanto a vida. Porém, não se trata do quão forte você pode atingir, se trata do quão forte você pode ser atingido e continuar seguindo em frente. É assim que a vitória é conquistada."

Rocky Balboa

RESUMO

O projeto Awareness tem como principal objetivo determinar se é possível estimar o quanto um pedestre está focado no tráfego de automóveis, enquanto utiliza um dispositivo móvel (smartphones, tablets, smartwatches). Utiliza uma simulação imersiva de realidade virtual do contexto de um pedestre em uma via, para realizar testes e coletar os dados necessários para construir um modelo computacional de foco perto de vias de tráfego de automóveis. Nesta simulação é necessário que existam entidades que apresentem diferentes elementos presentes nas vias, destacando-se os mais essenciais como automóveis e pedestres. Levando em conta o fato de que o comportamento destes elementos é altamente complexo, encontra-se a necessidade de ter na simulação entidades cujos comportamentos aproximem-se da complexidade encontrada no mundo real.

Objetivando sanar esta necessidade este trabalho propõe o uso do conceito da inteligência artificial de agente inteligente, que caracteriza-se por sistemas que podem decidir por si próprios o que eles precisam fazer para satisfazer seus objetivos pré-designados (WOOLDRIDGE, 2002). O foco é desenvolver um modelo de interação entre um agente e um ambiente de realidade virtual como o do projeto Awareness. Sendo assim este trabalho tem como objetivo propor uma arquitetura para a atuação de agentes em ambientes virtuais 3D, e realizar um estudo de caso com implementação da arquitetura no projeto Awareness.

Palavras-chave: agentes, simulação, atenção, foco, tráfego, automóveis.

ABSTRACT

The Awareness project main goal is to determine if its possible to estimate how much a pedestrian is focused on the car traffic while using a mobile device (smartphones, tablets, smartwatches). It uses a virtual reality immersive simulation of the context of a pedestrian in a road, to realize tests and collect the necessary data to build a computational model of focus near roads of car traffic. In this simulation it's necessary that exists entities that show different elements present in roads, highlighting the most essentials such as cars and pedestrians. Taking in account that the fact that the behavior of this elements it is of high complexity, there is the necessity of having in the simulation, entities with behaviors that approach the complexity found in the real world.

Aiming to solve this necessity this work proposes the use of the artificial intelligence's concept of intelligent agent, that is characterized by a system that can decide on its own what they need to do in order to satisfy its design goals (WOOLDRIDGE, 2002). The focus is to develop a model of interaction between an agent and a virtual reality environment such as the Awareness project environment. Therefore this work has as it’s goal to propose an architecture that allows agents to actuate on virtual 3D environments, and realize a study case with implementation of the proposed architecture on the Awareness project.

Keywords: agents, simulation, attention, focus, roads, cars

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

É a partir da observação da natureza dos computadores que pode-se facilmente notar: os computadores não pensam por si só. Computadores e seus programas funcionam baseados em diretivas bem definidas: se a porta está aberta você deve entrar; se a porta está fechada você deve abri-la e depois entrar. A tecnologia desenvolvida é muito boa em obedecer a esta diretivas, porém com o avanço da era da informação nota-se cada vez mais a necessidade de softwares que tomem decisões por si só. Podemos observar exemplos deste tipo de software já tomando forma em nosso dia-a-dia como é o caso do carro autônomo da empresa Uber (G1, 2016). É neste momento em que encaixa-se a inteligência artificial, a qual pode ser definida como o ramo da ciência da computação que se ocupa da automação do comportamento inteligente (LUGER, 2014).

Dentro dos diversos ramos da inteligência artificial, no contexto de tomada de decisão, temos o conceito de agentes. Pode-se definir agentes como sistemas capazes de decidir por si próprios o que eles precisam fazer para satisfazer seus objetivos pré-designados (WOOLDRIDGE, 2002). Uma das teorias de agentes é o modelo BDI: *belief-desire-intention* (em tradução livre: crenças-desejos-intenções), este modelo parte da idéia de que estes 3 pontos: crenças, desejos e intenções fazem parte fundamental do processo de tomada de decisão humano. Nesta arquitetura cada um dos 3 pontos desempenha um papel importante: as crenças são uma representação aproximada do estado atual do ambiente - é através desta representação que o agente decide o que é possível realizar ou não em um determinado momento. Baseada nas opções do que é possível realizar naquele momento, os desejos funcionam como motivador para as ações do agente. Uma vez determinadas as possíveis opções de ação através das crenças, escolhidas quais ações são interessantes para a função específica daquele agente através dos desejos, é então escolhido um conjunto de ações a serem tomadas para atingir os objetivos, ações estas que são representadas na forma de intenções.

Este trabalho tem como motivação inicial o projeto Awareness, que visa construir um modelo de atenção para usuários de dispositivos móveis enquanto trafegam em vias de automóveis com o objetivo de prever o nível de atenção de um usuário. Utilizando o modelo construído combinado com informações obtidas pelo próprio dispositivo móvel do usuário, torna-se possível alertar o usuário sobre situações possivelmente perigosas de forma congruente com o seu nível de atenção.

Para construir um modelo de atenção o projeto utiliza de um ambiente de imersão virtual CAVE, um sistema de simulação onde uma pessoa encontra-se em uma sala com diversos aparatos para criar uma total imersão como por exemplo imagens projetadas nas paredes ao seu redor [VRS, 2016]. No caso do projeto a simulação é a de um usuário utilizando um dispositivo móvel em uma via automotiva. Nesta simulação estão presentes os principais elementos que caracterizam-se como importantes para a construção do modelo tais como carros (conduzidos por humanos), pedestres, semáforos e distrações.

Um ambiente CAVE define-se por uma sala onde são posicionados vários projetores de forma a criar projeções nas paredes, cuja quantidade pode ser variável dependendo da necessidade da simulação, um dispositivo de entrada que varia da mesma forma e óculos estereoscópicos.

Visando a construção de uma simulação com maior fidelidade, encontra-se a necessidade de simular os motoristas e pedestres. Nota-se que o comportamento dos seres humanos é de alta complexidade, definido pelas mais diversas características como sentimentos, desejos, vontades e motivações. Sendo assim é necessário um paradigma de desenvolvimento que permita representar estes humanos. Levando em conta que o ambiente onde o agente encontra-se é de alta complexidade - cenário pouco comum no contexto de desenvolvimento de agentes - este trabalho propõe uma arquitetura para que agentes atuem em ambientes de complexidade, como por exemplo o ambiente CAVE anteriormente descrito.

1.1 OBJETIVOS

1.1.1 Objetivo geral

Este trabalho tem como objetivo geral propor uma arquitetura para agentes que permita a captura de informações e tomada de ações sobre ambientes virtuais.

1.1.2 Objetivos específicos

- Realizar uma pesquisa na literatura sobre o estado da arte da área de agentes no que diz respeito a atuação em ambientes virtuais;
- Realizar a implementação da arquitetura proposta de forma a tornar factível a implementação de agentes cujo ambiente esteja separado do framework do agente;

- Realizar um estudo de caso acerca da implementação realizada, modelando agentes para representar os pedestres na simulação imersiva do projeto Awareness.

2. REFERENCIAL TEÓRICO

Esta seção tem como objetivo criar um embasamento teórico no que diz respeito a agentes e os assuntos relacionados, sendo parte fundamental para o entendimento deste trabalho e também do primeiro objetivo específico deste trabalho.

2.1 AGENTE

(WOOLDRIDGE, 2002) descreve que um agente é um sistema computacional que está situado em algum ambiente e é capaz de realizar ações autônomas nesse ambiente visando cumprir seus objetivos de modelagem. Aprofundando-se na definição de agente de (WOOLDRIDGE; JENNINGS, 1995) que define agentes principalmente através de sua capacidade em tomar decisões, tendo como palavra chave autonomia: um agente deve ser capaz de tomar decisões alinhadas com seus objetivos sem a intervenção ou supervisão de humanos ou outros sistemas. Esta mesma definição sugere que agentes devem ser implementados através de conceitos-chave do processo de tomada de decisão humano como conhecimentos, crenças, intenções e obrigações.

(BONSON,2012) cita como exemplo de agente: um robô que explora uma área desconhecida a procura de um objeto, com atuadores e sensores para interagir com o ambiente ou então um agente jogador de xadrez com atuadores para mover as peças e sensores para compreender as jogadas do adversário.

Para um correto entendimento do contexto de agentes, faz-se necessário o entendimento da Teoria de Agentes. Visando defini-la (BONSON,2012) cita a definição de (WOOLDRIDGE; JENNINGS, 1995a): teorias de agentes são essencialmente especificações, que buscam definir aspectos tais como o que um agente é, e que propriedades ele deveria ter, e como pode-se representar e raciocinar sobre estas propriedades. É através da ótica proposta pela teoria de agentes, que algumas abordagens foram elaboradas, sobre como deve-se construí-los.

(BONSON,2012) descreve que atualmente a abordagem mais utilizada é a BDI (Belief-Desire-Intention) que envolve dois processos principais: decidir quais metas deseja-se atingir (deliberação) e como serão atingidas (meios-fins). Noções mentais humanas e uma perspectiva social são utilizadas na modelagem dos agentes. Os 3 conceitos básicos que compõem um agente BDI são:

- Crenças (Belief): Conjunto de crenças que o agente possui sobre o ambiente. É realizando uma observação do ambiente o qual está inserido, que o agente pode adquirir uma série de percepções do mesmo e com base nelas cria seu conjunto de crenças.

- Desejos: Conjunto de estados do ambiente que o agente deseja “atingir”.

- Intenções: Sequência de ações que o agente se compromete a executar para atingir seus objetivos. Uma vez comprometido com uma intenção, o agente dará início à execução sequencial das ações daquela intenção, onde cada uma das ações consiste em de algum modo atuar sobre o ambiente o qual o agente está inserido.

Outra abordagem amplamente estudada é a de agentes reativos. De acordo com (WOOLDRIDGE, 2002) a abordagem reativa mais conhecida é a de subsunção, desenvolvida por Rodney Brooks. Nesta abordagem o processo de tomada de decisão é realizado através de um conjunto de comportamentos, no qual cada comportamento pode ser considerado como uma função que continuamente mapeia as percepções de entrada em uma ação a ser executada. Outra caracteríıstica importante da arquitetura de subsunção é que vários comportamentos podem ser ativados ao mesmo tempo. (BONSON,2012) cita algumas vantagens da abordagem reativa: simplicidade, economia, tratabilidade computacional, robustez e elegância. E também algumas desvantagens como o fato de os agentes se basearem em informação local, sendo difícil tomarem decisões levando em consideração informações não-locais.

2.2 LINGUAGENS E ARQUITETURAS DE AGENTES

Uma linguagem de agente é um sistema que permite programar e executar agentes, e que pode utilizar princípios estabelecidos pelas teorias de agentes (WOOLDRIDGE; JENNINGS, 1995a).

A implementação de agentes também se faz possível através da utilização de uma arquitetura de agente, ao invés da utilização de uma linguagem. (WOOLDRIDGE; JENNINGS, 1995) propõe que a distinção entre linguagens e arquiteturas de agentes é de certa forma artificial, já que algumas das arquiteturas também podem ser consideradas linguagens. Boa parte do interesse em linguagens de agentes é motivado pelo conceito de programação orientada a agentes. Este paradigma baseia-se na idéia de de programar agentes nos termos e noções descritos pela teoria de agentes para que seja possível

construir agentes nos mesmos termos que usamos para descrever o comportamento humano: através de lógica convencional.

Dentre as linguagens e arquiteturas citadas por (BONSON,2012) destacam-se para este trabalho:

- AgentSpeak (FISHER et al., 1993): AgentSpeak(L) é uma linguagem de alto-nível, e definida para ser uma extensão da programação lógica que incorpore crenças, eventos, objetivos, ações, planos e intenções. O comportamento dos agentes é dirigido pelos planos escritos em AgentSpeak. Embora as crenças, desejos e intenções do agente não sejam representadas explicitamente como lógica modal, elas podem ser formalmente atribuídas ou checadas dada uma representação do estado atual do agente.

- Jason (BORDINI; HUBNER, 2007): Jason é um interpretador de uma versão melhorada da linguagem AgentSpeak(L), que inclui comunicação baseada na teoria de atos de fala. JASON é implementado em Java, sendo multi-plataforma e código-aberto.

- Jadex (PASSARELLA, 2016): O framework Jadex possui uma representação explícita dos objetivos e uma forma de integrar os mecanismos de determinar os objetivos. Por outro lado, o sistema respeita o estado da arte atual em relação a orientação a objetivos em engenharia de software, a fim de atrair não somente especialistas em inteligência Artificial, mas também desenvolvedores de software habilidosos. Por causa disso, o desenvolvimento de agentes é baseado em Java e XML, e apresenta suporte a aspectos de engenharia de software, como módulos reusáveis e ferramentas de desenvolvimento. Analisando a arquitetura de Jadex, um agente é uma caixa preta que recebe e envia mensagens.

- 2APL (DASTANI, 2007): É um framework que trata também de sistemas multiagente, contanto possui uma clara divisão entre a programação dos agentes individualmente e da arquitetura multi-agente. No nível de agente individual, os agentes são implementados em termos de crenças, objetivos, ações, planos, eventos e três diferentes níveis de regras. As crenças e objetivos dos agentes são implementados de forma declarativa, enquanto planos e a interface para o ambiente é implementada de forma imperativa. A parte declarativa suporta a implementação de raciocínio e

atualizações do estado mental dos agentes. Já a parte imperativa facilita a implementação de planos, do fluxo de controle.

- TouringMachines: Consiste de três camadas produtoras de atividades (camada reativa, camada de planejamento, camada de modelagem) que utilizam diferentes níveis de abstração para gerar continuamente sugestões sobre qual ação o agente deve tomar, e um sistema de controle,que decide qual camada terá controle sobre as ações do agente.

2.3 AGENTES EMBARCADOS

Embodied agent classifica-se como um agente que interage com o ambiente através de um corpo físico (PERIN,2017). Este ambiente pode ser real ou virtual. Um exemplo de embodied agent em um ambiente real, seria um robô controlado por um agente, e utilizaria os sensores físicos deste robô para desenvolver suas crenças. Em um ambiente virtual, como por exemplo uma simulação gráfica, um embodied agent poderia ser um humano controlado por um agente, e as mãos e pernas poderiam ser os atuadores deste agente.

2.4 AMBIENTE

Ambiente é o local onde o agente está situado, dado que agentes podem ser utilizados nos mais diversos contextos (PERIN,2017) cita a definição de (RUSSEL; NORVIG; 2004) que caracteriza os ambientes como:

- Completamente observável ou parcialmente observável: um ambiente é completamente observável se os sensores do agente podem obter de forma precisa, completa e atualizada as informações do estado do ambiente. Caso existam ruídos ou imprecisões, o ambiente é parcialmente observável;

- Estático ou dinâmico: um ambiente estático permanece inalterado até que o agente execute alguma ação sobre ele. Um ambiente dinâmico pode mudar enquanto o agente está deliberando sua próxima ação;

- Discreto ou contínuo: a separação entre discreto e contínuo pode ser feita a partir dos estados do ambiente, das percepções e ações do agente e do tempo. Um jogo

de xadrez, por exemplo, é discreto pois apesar de possuir um grande números de estados, este número é finito.

2.5 CONCLUSÃO

Uma vez feito o embasamento sobre agentes, fica claro o papel da teoria de agentes no desenvolvimento dos mesmos, e também que as diferentes teorias apresentadas possuem vantagens e desvantagens uma sobre a outra, fazendo assim com que a escolha depende do tipo de problema que quer resolver-se com agente. O problema que quer resolver-se também determina em que tipo de ambiente o agente está inserido, fazendo do ambiente também um critério importante.

3. TRABALHOS RELACIONADOS

Nesta seção serão estudados trabalhos que possuem um objetivo semelhante ao deste trabalho ou que utilizam dos conceitos que estão relacionados com o referencial teórico anteriormente descrito.

3.1 SIMULAÇÃO DE MULTIDÕES PARA RESPOSTA DE EMERGÊNCIA USANDO AGENTES BDI BASEADO EM REALIDADE VIRTUAL

O trabalho de (SHENDARKAR;VASUDEVAN,2008) tem como objetivo simular um cenário de emergência, como por exemplo o cenário de uma cidade onde ocorre uma explosão.

Nesta simulação obviamente estão presentes pessoas, que irão se comportar de diversas maneiras como buscar uma saída, buscar um policial ou ajudar outras pessoas. Dito isto o trabalho em questão faz uso de agentes BDI para simular o comportamento das pessoas no cenário de emergência. O uso de agentes se faz importante nesse caso para que os humanos simulados possuam uma complexidade suficiente para representar o comportamento humano.

(SHENDARKAR;VASUDEVAN,2008) esclarece que os sistemas mais simples de simulação atuais são baseados na idéia de que os indivíduos têm comportamento homogêneo o que impossibilita uma boa representação do comportamento de humanos. Ficando assim clara a necessidade da utilização de agentes.

(SHENDARKAR;VASUDEVAN,2008) descreve as características gerais de um agente BDI e cita suas vantagens no contexto do trabalho, se comparado a outras arquiteturas, justificando a escolha do paradigma:

- O paradigma BDI é baseado em lógica de senso comum, onde os conceitos do paradigma podem ser facilmente traduzidos para a linguagem que as pessoas usam para descrever seu raciocínio e ações no dia-a-dia.
- É um framework relativamente maduro e possui exemplos de sistemas de média-larga escala que utilizam-o.

A modelagem de agentes BDI é desenvolvida no software AnyLogic que possui uma biblioteca chamada AgentBase que facilita o desenvolvimento de agentes. Os modelos do

AnyLogic podem ser utilizados junto com bibliotecas mais sofitsticadas para diferentes tipos de agentes.

O ambiente do trabalho em questão fica caracterizado como completamente observável pois se trata de uma simulação virtual, dinâmico pois o ambiente sofre alterações constantemente (ex: fogo se espalhando) e contínuo.

3.2 INTEGRAÇÃO DE APRENDIZADO EM AGENTES BDI: ARQUITETURA PARA PROCESSAMENTO DE PERCEPÇÕES DE EMBODIED AGENTS

O trabalho de (PERIN, 2017) visa o desenvolvimento de uma arquitetura de agente BDI que integre aprendizado de máquina visando unir os componentes pró-ativo e reativo de forma a flexibilizar o processamento das percepções de embodied agents.

No trabalho em questão será utilizado BDI como componente pró-ativo do agente e o aprendizado de máquina como componente reativo, de forma a aprender a tratar os eventos incomuns aos quais o agente pode se deparar.

(PERIN, 2017) informa que a arquitetura desenvolvida irá passar por um estudo de caso no projeto Awareness, projeto já descrito anteriormente, sendo assim o ambiente se caracteriza como completamente observável, dinâmico e contínuo.

3.3 DESENVOLVIMENTO DE UMA ABORDAGEM DE COOPERAÇÃO EM SISTEMAS MULTIAGENTES

Já o trabalho de (PASSARELLA, 2016) realiza um estudo sobre abordagens para cooperação de agentes, em sistemas multi-agentes. Neste trabalho foi elaborado um estudo de caso em cima de um exemplo que é basicamente composto por carros que cooperam entre si em um cruzamento sem necessidade de semáforos.

(PASSARELLA, 2016) utiliza a teoria de agentes BDI e Jason como arquitetura. O objetivo é construir uma camada de agentes racionais que sobreponha a infraestrutura mediadora e permita a construção de agentes inteligentes usando os conceitos de engenharia de software (POKAHR, BRAUBACH, LAMERSDORF, 2005).

(PASSARELLA, 2016) informa sobre o framework utilizado no desenvolvimento do trabalho: JaCaMo é um framework para programação de sistemas multiagentes combinando três tecnologias: Jason, para programação de agentes autônomos; Cartago,

para programação de artefatos de ambiente; e MOISE para programação de organizações multiagentes.

O ambiente onde são executados os testes é uma simulação simples 2D desenvolvida na ferramenta JaCaMo.

3.4 TEORIA DE SUBSUNÇÃO APLICADA PARA CONTROLE DE VOO USANDO COMPOSIÇÕES DE ROTAÇÃO

(KRISTIANSEN, 2016) propõe a utilização da teoria de subsunção para o desenvolvimento de agentes capazes de controlar veículos aéreos não tripulados em um território urbano evitando colisões com o chão e com outros obstáculos.

Para atingir tal objetivo é utilizada a teoria de subsunção onde múltiplas tarefas são organizadas em uma hierarquia de forma que as tarefas de nível não primário fiquem submissas às de nível primário. Este tipo de hierarquia permite que sempre busque-se atingir a tarefa primária e assim que esta estiver completa busca-se atingir as de mais baixo nível. Segundo (KRISTIANSEN, 2016) isto permite que comportamentos complexos sejam decompostos em várias tarefas mais simples.

Ao contrário dos trabalhos anteriormente citados, o trabalho de (KRISTIANSEN, 2016) propõe um agente que atua no mundo real, caracterizando assim o ambiente do trabalho como parcialmente observável pois os sensores não podem captar as informações do mundo real com total precisão, dinâmico e contínuo.

3.5 AGENTES BDI EM SITUAÇÕES SOCIAIS

(GAUDOU, 2016) fornece um completo trabalho sobre a atuação de agentes em situações sociais. O trabalho informa que agentes BDI podem trazer importantes benefícios para simulações sociais, dependendo do objetivo da simulação, dos requisitos dos agentes, a escala de observação e o campo de aplicação.

Algumas capacidades fornecidas pela teoria BDI se destacam: adaptabilidade, robustez, programação abstrata e a habilidade de explicar o comportamento (GAUDOU, 2016) .

É Informado também que BDI não cabe em simulações baseadas em agentes nas seguintes situações:

- Campo de aplicação: BDI não cabe para modelar entidades simples como bactérias, mas é ideal para modelar humanos;
- Objetivo da simulação: BDI não cabe quando a simulação requer uma representação com alta precisão devido ao nível de complexidade da implementação;
- Limitações de escalabilidade: BDI não cabe por exemplo em jogos de tempo real.

(GAUDOU, 2016) propõe que apesar dos grandes benefícios fornecidos pela teoria BDI e da falta de boas razões para não utilizá-la, ela não é amplamente utilizada em simulações devido a falta de ferramentas apropriadas para a atuação de agentes BDI nas mesmas.

Quadro 1 - RESUMO DOS TRABALHOS CORRELATOS

Trabalho	Tipo de ambiente	Teoria	Propósito do sistema /Propósito do agente	Linguagem/ Arquitetura
Simulação de multidões para resposta de emergência usando agentes BDI baseado em realidade virtual (SHENDARKAR; VASUDEVAN,2008)	Completamente observável; dinâmico; contínuo	BDI	Simular comportamento de humanos em multidões	AgentBase
Integração de aprendizado em agentes bdi: arquitetura para processamento de percepções de embodied agents (PERIN, 2017)	Completamente observável; dinâmico; contínuo	BDI	Integração de agentes com aprendizado de máquina / Simulação de pedestres	JASON

Desenvolvimento de uma abordagem de cooperação em sistemas multiagentes (PASSARELLA, 2016)	Completamente observável; dinâmico; discreto	BDI	Cooperação de agentes usando planos compartilhados / carros cooperando entre si para que não haja necessidade de semáforos	JaCaMo
Teoria de subsunção aplicada para controle de voo usando composições de rotação (KRISTIANSEN, 2016)	Parcialmente observável; dinâmico; contínuo	Subsunção	Controle de voo de veículos aéreos não tripulados	Não definido
Agentes BDI em situações sociais (GAUDOU, 2016)	Não definido	BDI	Não definido	Não definido

3.6 CONCLUSÃO

Dado os trabalhos correlatos estudados neste capítulo fica claro que em contexto similares ao deste trabalho é frequentemente usado BDI e que as necessidades da simulação podem ser suportadas pela teoria. Observando especificamente o segundo trabalho, deixa clara a semelhança do propósito dos agentes, e que o framework JASON

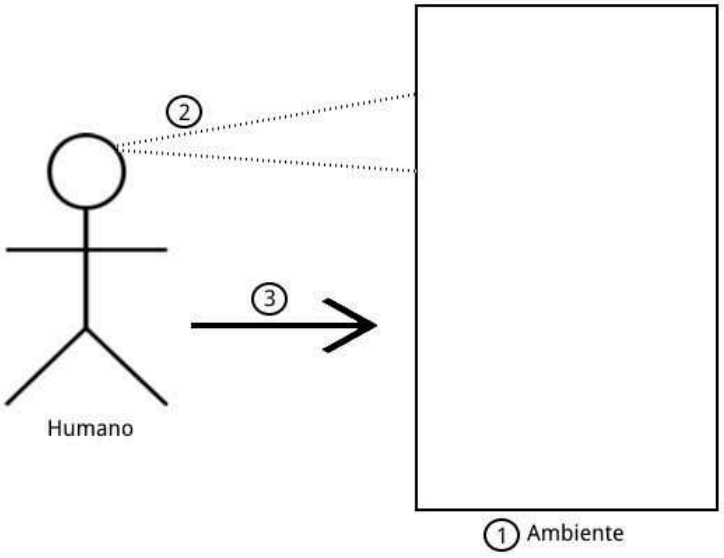
deve ser considerada como a mais provável escolha para o desenvolvimento da arquitetura proposta neste trabalho.

É válido salientar também que o último artigo analisado reforça a motivação deste trabalho, comunicando a ausência de ferramentas apropriadas para integração de agentes BDI em simulações. Com isso conclui-se o primeiro objetivo específico deste trabalho, que consiste em realizar uma pesquisa na literatura sobre o estado da arte da área de agentes no que diz respeito a atuação em ambientes virtuais.

4 DESENVOLVIMENTO

Este trabalho tem como objetivo principal propor uma arquitetura para agentes que permita a captura de informações e tomada de ações sobre ambientes virtuais 3D. Dado este objetivo, faz-se necessário um entendimento sobre o que a teoria de agentes propõe sobre como agentes e seus ambientes devem se relacionar. A teoria de agentes propõe um tipo de relacionamento ambiente x agente que conceitualmente se assemelha ao modo como os humanos interagem com o mundo:

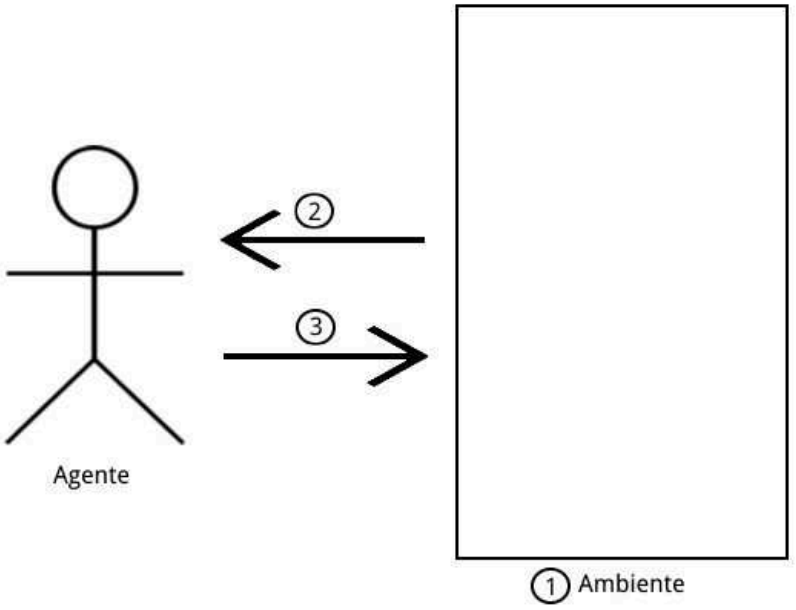
Figura 1 - RELACIONAMENTO HUMANO X AMBIENTE



Na imagem podemos observar o ambiente (número 1) que é realmente o meio em que o humano está inserido, o mundo real. O humano é responsável por observar o seu ambiente, e captar informações sobre ele (número 2): esta tarefa é realizada através dos sentidos do humano, como por exemplo a visão. O humano também interage com o ambiente atuando sobre ele (número 3) esta atuação pode ser mover um objeto com as mãos, por exemplo. Neste trabalho descreve-se este tipo de arquitetura como uma arquitetura de ambiente passivo.

Porém, observando o modo como as arquiteturas e linguagens de agentes como por exemplo JASON, JaCaMo, JADEX e 2APL funcionam de um modo geral, a interação agente x ambiente ocorre através de um fluxo diferente:

Figura 2 - RELACIONAMENTO AGENTE X AMBIENTE ATIVO



Na imagem pode-se observar que não existe uma “observação” do ambiente feita pelo agente, e sim uma seta que sai dele (número 1) para o agente. Esta seta representa o ambiente alimentando o agente com as informações que o agente irá precisar para seu fluxo de raciocínio. Ou seja, neste tipo de relacionamento o ambiente assume totalmente a responsabilidade de manter o agente atualizado com as informações que ele precisa, isto faz com que o ambiente precise atualizar os agentes com as modificações que ocorreram no ambiente - até quando os agentes não precisam de atualizações sobre o mesmo. Neste trabalho descreve-se este tipo de arquitetura como uma arquitetura de ambiente ativo. Este fluxo onde o ambiente alimenta o agente pode ser funcional e prático para alguns tipos de trabalhos com agentes, principalmente aqueles onde o ambiente fica dentro do próprio framework do agente, estes trabalhos em sua maioria são simulações simples. Porém quando amplia-se o contexto de atuação de agentes, como por exemplo agentes que atuam no mundo real (através de robôs, por exemplo) ou então em simulações onde o framework do ambiente roda em outra plataforma, fora do framework do agente isto torna-se um problema, pois conceitualmente não faz sentido “o mundo real” alimentar o agente com informações, e sim faz sentido o agente observar e angariar as informações que são de seu interesse. É exatamente neste ponto que contextualizam-se as idéias propostas por este trabalho.

A princípio esta questão tem uma implicação apenas conceitual, mas observando na prática o tipo de relacionamento que as arquiteturas e linguagens utilizam nota-se que

também existe uma implicação prática: em um cenário onde agentes atuariam no mundo real, não existe uma forma do mundo real alimentar o agente com informações, e seria necessário o desenvolvimento de um software separado que fosse responsável por observar o mundo real e alimentar o agente com as informações. O mesmo tipo de problema ocorreria em ambientes virtuais, cujo framework do ambiente é separado do framework do agente.

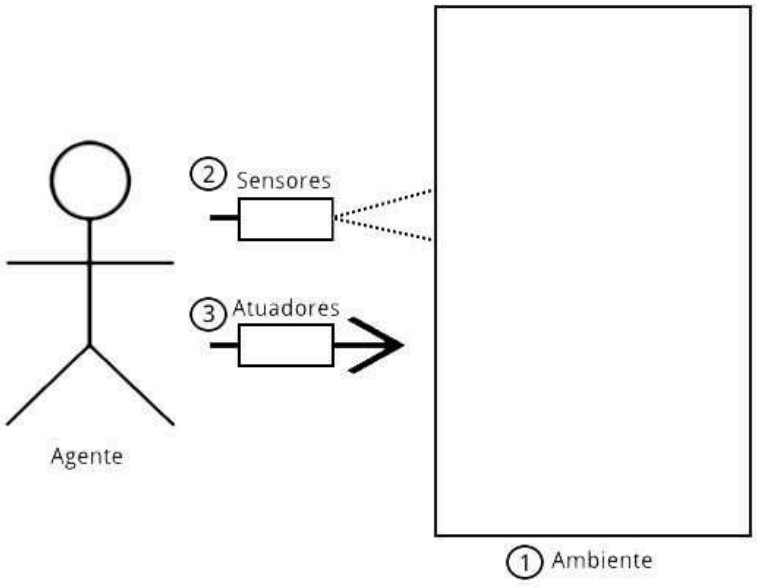
Observa-se que em casos onde tem-se um ambiente mais simples como em casos de interface gráfica muito simples (ex: um tabuleiro de jogo da velha) ou mesmo sem interface gráfica existe uma vantagem prática de ter este tipo de associação: a simplicidade na implementação. Já que o ambiente é simples e pode ser construído com facilidade no próprio framework dos agentes, é mais fácil construir as percepções do agente dentro do próprio ambiente e alimentá-los com elas.

Todavia outras implicações práticas negativas deste tipo de relacionamento são encontradas no código da arquitetura do agente e do ambiente:

- Falha na modelagem conceitual: Uma das atividades realizadas pelo ambiente consiste em processar a si mesmo de forma a construir as percepções de cada agente individualmente e alimentá-los com elas. Esta atividade claramente não é de sua responsabilidade, pois não é papel do ambiente alimentar o agente com informações e muito menos entender detalhes da construção das percepções dos agentes.
- Falha no encapsulamento: O encapsulamento na programação orientada a objetos consiste em ocultar detalhes do funcionamento interno de um determinado componente de forma que outros componentes que utilizam-o não precisem ter conhecimento do seu funcionamento para utilizá-lo. No caso da arquitetura de agentes temos a necessidade de expor detalhes da implementação da arquitetura de forma que o ambiente possa se comunicar com ela.

Visando sanar esta problemática este trabalho propõe o seguinte fluxo de relacionamento entre agente e ambiente:

Figura 3 - RELACIONAMENTO AGENTE X AMBIENTE PASSIVO



Através da figura 3 observa-se que o fluxo é sempre na direção do ambiente, isto se dá pois o ambiente desempenha um papel totalmente passivo (caracterizando esta arquitetura como uma arquitetura de ambiente passivo), se atendo totalmente apenas às suas funções nativas, deixando que o agente atue e observe-o da maneira e no tempo adequados. Desta maneira respeita-se o conceito de encapsulamento no código. Vendo a imagem de um ponto de vista mais técnico pode-se observar também a presença de dois novos componentes: os sensores (número 2) e atuadores (número 3). Estes componentes são os responsáveis por desacoplar do agente os comportamentos de observar e atuar sobre o ambiente, criando assim uma arquitetura mais flexível e dinâmica que, assim como o ambiente, respeita o conceito de encapsulamento.

O sensor do agente é responsável por implementar a maneira como ele observa o ambiente, implementando os detalhes da “comunicação” com o ambiente, e exercendo as operações necessárias sobre a observação realizada, de modo a interpretar os dados e retorná-los em forma de crenças para o agente. O atuador, da mesma maneira, precisa realizar a mesma comunicação com o ambiente e é responsável por implementar os detalhes de como o agente deve atuar sobre ele.

Sendo assim faz-se necessário um planejamento das diversas fases que envolvem o desenvolvimento devido a sua complexidade. O primeiro passo do desenvolvimento consiste em escolher qual abordagem de agentes que será utilizada, dentre as que já foram

elencadas neste trabalho. Um estudo mais aprofundado sobre as aplicações práticas de cada abordagem é necessário para identificar qual melhor se aplica às questões deste trabalho. Uma vez escolhida a abordagem será necessário escolher a arquitetura que será utilizada para a implementação, da mesma maneira é necessário um estudo das questões mais práticas e técnicas da arquitetura para saber qual se adapta de forma mais adequada. O próximo passo é realizar a implementação da arquitetura proposta, que torna factível a captura de informações e tomada de ações sobre ambientes virtuais. O último passo é um estudo de caso acerca da implementação construída mostrando que a arquitetura funciona. Sumarizando, o desenvolvimento se divide em:

1. Escolha da abordagem de agentes;
2. Escolha da arquitetura de agentes;
3. Implementação da arquitetura para atuação em ambientes virtuais 3D;
4. Estudo de caso.

4.1 ESCOLHA DA ABORDAGEM DE AGENTES

A abordagem de agentes, atendo-se a parte teórica do assunto, define quais teorias o agente deve seguir, define os componentes teóricos que constroem o agente. Pode-se comparar a abordagem de agentes com os paradigmas de linguagem de programação: o paradigma define os padrões que as linguagens que implementam os paradigmas devem seguir. A escolha da abordagem correta é fundamental, pois torna-se inviável a mudança de abordagem uma vez que a implementação está em um estágio avançado, pois a mudança de abordagem iria implicar na mudança da arquitetura, e na mudança total da forma como os agentes são construídos.

Existem algumas abordagens propostas pela teoria de agentes sobre como o desenvolvimento de agentes deve ocorrer. Uma das abordagens estudadas neste trabalho é a abordagem reativa de subsunção. Esta abordagem consiste em descrever o agente através de diversos comportamentos que atuam em conjunto para atingir o objetivo do mesmo. Esta arquitetura tem como algumas de suas principais vantagens a simplicidade, isto se dá pois é possível decompor comportamentos complexos em um conjunto de comportamentos de menor complexidade. Todavia observa-se também como desvantagem a fraca adaptabilidade da estrutura, já que comportamentos complexos são descritos em uma série de comportamentos menores, isto torna difícil modificar comportamentos já existentes ou adicionar novos. Também foi possível observar através dos trabalhos correlatos estudados na seção 3 que este tipo de arquitetura é mais adequada para cenários onde o agente tem de ter um comportamento mais reativo como o de um

controlador de veículo aéreo, como foi o caso estudado. Observa-se que este tipo de cenário é totalmente diferente do cenário abordado neste trabalho.

Outra abordagem é a BDI propõe como principais componentes teóricos a definição de agentes através de crenças, desejos e intenções. Esta abordagem destaca-se claramente como a mais adequada para este trabalho devido aos seguintes pontos:

- Flexibilidade: BDI permite uma alta flexibilidade e facilidade de alteração dos agentes implementados, pois a sua definição é feita através dos componentes teóricos, ou seja, não é necessário nenhum código “fixo” na arquitetura do agente para definir os comportamentos do mesmo. Já que os comportamentos ficam encapsulados, torna-se muito fácil a adição, remoção ou alteração dos mesmos;
- Comportamento enriquecido: A abordagem escolhida parte da idéia de que os três componentes (crenças, desejos e intenções) são parte fundamental do processo de decisão humano, e simulando os três pode-se ter um comportamento de alta complexidade, com certeza não tão complexo como o humano, mas com representatividade suficiente. Sendo assim, se a intenção do agente é simular por exemplo um humano, ou outra entidade de comportamento complexo, o BDI faz-se uma ótima escolha.
- Utilização em trabalhos similares: Levando em conta o objetivo deste trabalho de tornar factível a atuação de agentes em ambientes virtuais, pode-se induzir que em geral os agentes implementados através desta arquitetura provavelmente serão agentes que representam humanos, ou pelo menos entidades com complexidade similar. Sendo assim analisando os trabalhos correlatos estudados na seção 3 e comparando os contextos dos mesmos com o deste trabalho, fica evidente que neste tipo de ambiente a arquitetura BDI é adequada.

4.2 ESCOLHA DA LINGUAGEM/ARQUITETURA DE AGENTES

Uma vez escolhida a abordagem de agentes, abre-se um leque de escolhas das linguagens e arquitetura existentes que implementam essa abordagem. Pode-se comparar uma arquitetura/linguagem de agentes com uma linguagem de programação: enquanto Java ou C++ implementam o paradigma de orientação a objetos, arquiteturas como JASON e Jadex implementam a abordagem BDI.

Sendo assim faz-se necessário a escolha dentre as linguagens/arquiteturas elencadas nas seções 2 e 3 deste trabalho que implementam a abordagem BDI, que são:

- 2APL: É uma linguagem de programação orientada a agentes, onde define-se agentes através de crenças, objetivos, ações, planos, eventos e regras. Tem como um de seus principais objetivos facilitar o desenvolvimento de sistemas multi-agente, sendo assim fornece um arcabouço de funcionalidades em sua linguagem para cumprir este objetivo.
- AgentSpeak: É uma linguagem de programação orientada a agentes de alto nível onde define-se agentes através de crenças, eventos, objetivos, ações, planos e intenções.
- JASON: É uma arquitetura de agentes, que fornece estrutura para implementação de agentes em variados contextos. Nesta arquitetura os agentes são definidos através de uma versão estendida da linguagem de agentes AgentSpeak. A arquitetura JASON inclui um interpretador para esta linguagem e fornece toda a estrutura de forma a facilitar o desenvolvimento de agentes.
- JaCaMo: É uma arquitetura de agentes para o desenvolvimento de sistemas multi-agente. O JaCaMo é uma combinação de três outras tecnologias: JASON, Cartago[1] e Moise[2]. Cartago é uma infra-estruturada para o desenvolvimento de ambientes para sistema multi-agente baseada no modelo Agentes & Artefatos. Moise é um modelo para construção de organizações em sistemas multi-agente.
- JADEX: É uma arquitetura para o desenvolvimento de agentes baseado em componentes. Os agentes são definidos através de classes Java e arquivos XML. JADEX fornece uma boa estrutura no que diz respeito a engenharia de software.

Fazendo um comparativo entre as linguagens e arquiteturas acima, foi possível chegar a conclusões sobre cada uma delas. A linguagem 2APL tem seu principal foco no desenvolvimento de sistemas multi-agente, o que não considera-se essencial para este trabalho. A linguagem AgentSpeak se enquadra como uma boa opção, porém existe a arquitetura JASON, que só não implementa uma versão ainda mais poderosa da AgentSpeak como também oferece outras vantagens no que diz respeito ao desenvolvimento de agentes. A arquitetura JaCaMo oferece uma série de pontos positivos devido à sua combinação de tecnologias, porém as tecnologias Cartago e Moise são focadas para o desenvolvimento de sistemas multi-agente, que como já citado não é o foco deste trabalho. A arquitetura JADEX também classifica-se como uma boa opção já que

[1] http://cartago.sourceforge.net/
[2] http://moise.sourceforge.net/

fornece bom suporte para o desenvolvimento de agentes e implementa boas práticas de programação orientada a objetos.

Levando em conta estas considerações, faz-se necessário um comparativo entre as duas arquiteturas consideradas mais indicadas para este trabalho: JASON e JADEX. É através desta comparação que alguns pontos muito positivos foram identificados sobre a arquitetura JASON, tornando-a a mais indicada para o contexto deste trabalho:

- Agentes descritos em linguagem interpretada: Os agentes da arquitetura JASON são descritos em arquivos individuais chamados .ASL, nestes arquivos os agentes são descritos em termos de BDI através da linguagem AgentSpeak que possui sintaxe e semântica específica para o contexto de agentes. Facilitando assim o desenvolvimento, interpretação, modificação e a manutenção dos agentes.
- Grande abstração: Devido a arquitetura e modo como os agentes são implementados o desenvolvedor não tem de se preocupar com comportamentos específicos da estrutura da arquitetura e nem com comportamentos do motor BDI. O desenvolvedor tem apenas de se preocupar com a correta definição das crenças, desejos e intenções do agente.
- Utilização em trabalhos similares: Assim como a abordagem da arquitetura JASON (BDI) a própria arquitetura JASON é utilizada em trabalhos com o contexto similar ao deste, como elencam os trabalhos correlatos, formando assim um bom indicativo da congruência da arquitetura com os objetivos deste trabalho.

4.2 IMPLEMENTAÇÃO DA ARQUITETURA PARA ATUAÇÃO DE AGENTES EM AMBIENTES VIRTUAIS

Alinhado com o terceiro objetivo específico deste trabalho deu-se início a implementação da arquitetura proposta. Inicialmente foi necessário uma pesquisa sobre a arquitetura JASON, buscando compreender como é realizada execução de uma simulação em JASON e onde pode-se encontrar o código fonte.

O JASON é disponibilizado através de uma biblioteca em formato *.jar* e o download da mesma pode ser feita no site da arquitetura[3]. Também é possível realizar o download do código fonte no mesmo site. Para rodar uma aplicação em JASON é disponibilizado um

[3] http://jason.sourceforge.net/

plugin[4] para a plataforma de desenvolvimento Eclipse[5], também no site da arquitetura. Este plugin permite executar um projeto JASON diretamente da interface gráfica do Eclipse. Feito o download de ambos deve-se executar a biblioteca em formato *.jar* para que sejam feitas as configurações necessárias para a execução. Executando a biblioteca pode-se alterar as seguintes configurações:

- Localização da biblioteca: Define onde está localizada a biblioteca a ser utilizada pelo plugin do Eclipse para executar um projeto JASON;
- Localização da biblioteca Java: Define a localização da biblioteca Java a ser utilizada pelo JASON;
- Quais infra-estruturas podem ser utilizadas. Pode-se escolher diferentes infra-estruturas para rodar um projeto JASON, como descrito no parágrafo a seguir;
- Outras configurações técnicas de menor importância;

Um projeto JASON consiste de uma série de arquivos com responsabilidades bem definidas:

- Definição-base do projeto: Consiste na definição de uma série de configurações em um arquivo chamado *nomedoprojeto.mas2j*, destacando-se dentre elas:
  - Tipo de infra-estrutura: Pode-se utilizar uma estrutura centralizada, distribuída (onde diferentes agentes podem estar localizados em diferentes computadores) dentre outras disponibilizadas. Este trabalho foca na utilização de uma arquitetura centralizada;
  - Agentes: Define cada um dos agentes através de um nome;
  - Ambiente: Define uma classe *.java* para representar o ambiente dos agentes e adicionar comportamentos customizados ao mesmo;
  - Diretório: Define em qual diretório do projeto a especificação dos agentes se encontra;
- Definição dos agentes: Cada um dos agentes é definido em um arquivo individual. Nestes arquivos todos os componentes, como crenças, objetivos e planos de um agente são definidos através de uma versão estendida da linguagem AgentSpeak. Este arquivo será interpretado pela arquitetura

[4] http://jason.sourceforge.net/mini-tutorial/eclipse-plugin/
[5] http://www.eclipse.org/

JASON de forma a realizar o comportamento do agente através do motor BDI.

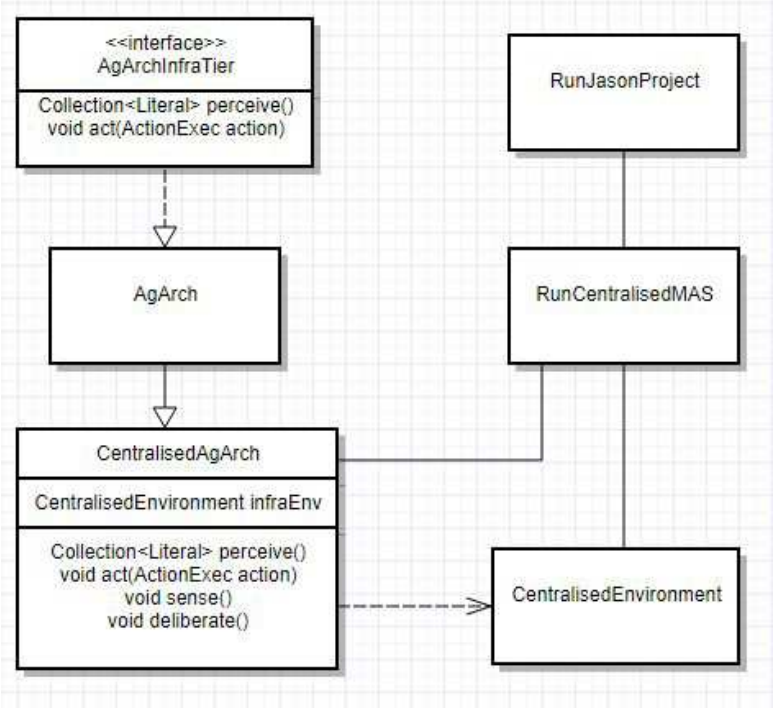
- Definição do ambiente: Na classe do ambiente define-se o ambiente dos agentes, o ambiente é responsável por: controlar a execução da simulação, definir as percepções de cada agente, implementar a execução das ações dos agentes, e também pode-se definir uma interface gráfica.

Feita essas observações iniciais, deu-se inicio á um estudo no código-fonte do JASON. O download do código-fonte pode ser realizado no repositório[6] da arquitetura. O código-fonte vem configurado para ser utilizado em conjunto com a ferramenta Gradle que fornece diversas facilidades para a modificação/desenvolvimento da arquitetura, dentre elas:

- Gerar o *.jar* da arquitetura com as modificações realizadas.
- Gerar a documentação a partir do código.
- Gerar um projeto Eclipse do código-fonte pronto para ser utilizado.

Gerando o projeto Eclipse através do Gradle e estudando-o pôde-se realizar uma série de observações sobre a arquitetura.

Figura 4 - DIAGRAMA DE CLASSES SIMPLIFICADO DA ARQUITETURA JASON



[6] http://github.com/jason-lang/jason

A arquitetura dos agentes JASON, no que diz respeito a este trabalho e pode ser visualizada de forma resumida na imagem acima, é definida em uma espécie de camadas estilo “cebola” onde é possível acoplar novas camadas ao fim de uma estrutura já existente, para assim adicionar novos comportamentos sem perder os já existentes. O ínicio desta estrutura em camadas se dá na classe *AgArch* que implementa a interface *AgArchInfraTier* que define os métodos que as camadas do agente devem implementar. Uma outra classe que segue nesta estrutura, logo abaixo da *AgArch* na hierarquia é a *CentralisedAgArch* que é responsável por implementar agentes em uma arquitetura centralizada (tipo de arquitetura necessária para este trabalho). Nesta classe podemos ver métodos que pertencem ao ciclo de raciocínio do agente definido pela abordagem BDI como:

- *sense():* Responsável por dar início ao processo que atualiza as crenças do agente;
- *deliberate():* Inicia o processo de deliberação do agente. Neste processo ele irá deliberar sobre os objetivos atuais dos agentes e a possível evolução de intenções para objetivos;

Também estão presentes nesta classe métodos que são chamados pelo ciclo de pensamento do agente como:

- *act(ActionExec):* Este método recebe como parâmetro um objeto do tipo *ActionExec,* que possui dentre seus atributos uma representação de uma ação a ser executada e o objetivo que deu origem a esta ação. Dada esta ação, o método irá executar um método *act(String, ActionExec)* da classe *CentralisedEnvironment* que representa o ambiente o qual os agentes estão inseridos, informando-a que ela deve executar esta ação com determinado agente;
- *perceive():* Irá apenas chamar *getPercepts(String)* também da classe *CentralisedEnvironment* informando-a o nome do agente, e aguarda como retorno as percepções do agente informado;

Dado este estudo inicial já é possível concluir que o JASON roda uma arquitetura de ambiente ativo, tendo o ambiente como responsável por produzir as percepções do agente através do método *getPercepts(String)*. Sendo assim, como esperado, se fez necessário uma modificação no fluxo de operação da arquitetura do agente, para torná-la uma arquitetura de ambiente passivo. O próximo passo então foi identificar como rodar o JASON

com as modificações a serem feitas. O código desenvolvido pode ser encontrado no repositório do Github[7].

Através dos estudos foi identificado que o plugin do Eclipse que executa o projeto JASON utiliza a classe *RunJasonProject* como ponto de partida para a execução do mesmo. Esta classe é responsável por receber como parâmetro a localização do arquivo de configuração *.mas2j,* buscar as configurações realizadas através da execução do *.jar* da arquitetura e dar início a execução através da criação da uma instância da classe *RunCentralisedMAS* (esta classe é utilizada pois neste trabalho utiliza-se uma arquitetura centralizada, outras classes são responsáveis por implementar os outros tipos de arquiteturas). Realizando um teste simples de modificação na classe *RunJasonProject* percebeu-se que a mudança não surtiu efeito, isto se deu pois esta classe na verdade utiliza o código presente no *.jar* da arquitetura para executar o projeto. Sendo assim é necessário recompilar o projeto gerando um novo *.jar* sempre que uma modificação for feita, para que seja possível executá-la. O processo de geração do *.jar* é facilitado através da ferramenta Gradle, sendo necessário apenas um comando para gera-lo. Uma vez gerado o novo *.jar* basta atualizar a configuração da localização do *.jar* e executar a classe *RunJasonProject.* Gerado o *.jar* foi realizado um novo teste simples e as modificações surtiram efeito com sucesso. Apesar da geração do *.jar* ser facilitada este modo de execução através de uma biblioteca externa dificultou muito o processo de estudo da arquitetura, já que não foi possível debugar a execução do mesmo.

Sendo possível realizar modificações e testá-las, o seguinte passo, visando tornar o ambiente passivo, foi desacoplar o ambiente da arquitetura do agente (*CentralisedAgArch)* que possui como um de seus atributos um objeto da classe *CentralisedEnvironment*, este objeto foi removido no processo de desacoplamento, junto com todas as linhas de código que utilizavam-no. Uma vez removidas as dependências ambos os métodos *act(ActionExec)* e *perceive()* param de funcionar, já que estes são diretamente dependentes do ambiente. Então foi realizado um teste simples, fazendo com que o método *perceive()* retornasse uma lista vazia de percepções e que o método *act(ActionExec)* não realizasse nada. Feito isso foi executado um projeto simples de teste, com um agente sem desejos para garantir que o motor continuava rodando mesmo sem novas percepções. Como esperado o motor rodou sem apresentar erros.

[7] https://github.com/hprandi/jason

Foi então dado início a implementação dos componentes atuador e sensor. Seguindo a idéia de que ambos os componentes não implementam um comportamento em si, apenas definem o padrão a ser seguido foi optado por definir interfaces com os nomes *Actuator* e *Sensor.* Com os respectivos métodos:

- *Actuator:* Representa o componente atuador. A classe que implementa a interface Actuator deve implementar um método com a assinatura *act(ActionExec)*, sendo assim será responsável por implementar todo o processo de execução de uma ação;
- *Sensor*: Representa um sensor, e a classe que implementa-a deve implementar o método de assinatura *perceive()* e fica responsável por implementar o processo de captação das percepções do agente e retorná-las em forma de coleção;

Além disso ficou decidido que estes componentes poderiam ser individuais por agente, ou seja, deveria ser possível definir para cada agente seus próprios sensores e atuadores. Para realizar a criação das instâncias dos componentes foi criado uma terceira interface chamada *AgentDefinition*, a classe que implementa esta interface deve realizar a implementação de três métodos: getName() que deve retorna o nome do agente, getActuator() que deve retornar uma instância da interface *Actuator* e o método *getSensor()* que deve retornar uma instância da interface *Sensor.*

Definido o modo como os componentes devem ser implementados o próximo passo foi decidir como esses componentes seriam injetados dentro da arquitetura do agente, representada pela classe *CentralisedAgArch.* Para isso foi necessário um novo estudo sobre a arquitetura do JASON buscando entender como é dado o processo de criação dos agentes a partir dos arquivos de definição *.asl*.

Foi iniciada uma busca objetivando localizar onde são criadas as instâncias da classe *CentralisedAgArch,* que representam a arquitetura de cada gente individualmente*,* para que fosse possível parametriza-lá com a instância do sensor e do atuador do respectivo agente. Foi identificado que o método responsável por criar as instâncias é o *createAgs()* da classe *RunCentralisedMAS.* Sendo assim foi criado um novo método nesta mesma classe chamado *loadDefinition(String),* que é responsável por buscar na raiz do projeto JASON uma classe com o nome do agente no formato *nomeDoAgente.java* e esta classe deve implementar a interface *AgenteDefinition*. Ou seja, para cada agente criado no projeto deve-se criar uma classe *.java* com o mesmo nome do agente, e esta classe ficará responsável por instanciar o atuador e o sensor do agente. Uma vez carregada a definição do agente através deste método o próximo passo foi criar dois parâmetros no construtor da

classe *CentralisedAgArch* para que esta possa receber o atuador e o sensor como parâmetros.

Tratando da classe *CentralisedAgArch* que agora recebe sensor e atuador como parâmetros, ambos os parâmetros foram alocados em atributos para que possam ser chamados quando necessários. A utilização do atuador se deu no método *act(ActionExec)* (que até o presente momento se encontrava vazio, devido ao desacoplamento do ambiente) de forma que agora o método tem apenas de chamar o método *act(ActionExec)* do atuador. A sua implementação se dá de forma simples já que agora a responsabilidade de implementar como a ação é executada cabe apenas ao atuador. Sendo assim o método *act(ActionExec)* ficou implementado da seguinte forma:

```
@Override
public void act(ActionExec action) {
    this.actuator.act(action);

    if (this.masRunner.getEnvironmentInfraTier() == null) {
        action.setResult(true);
        this.actionExecuted(action);
    }
}
```

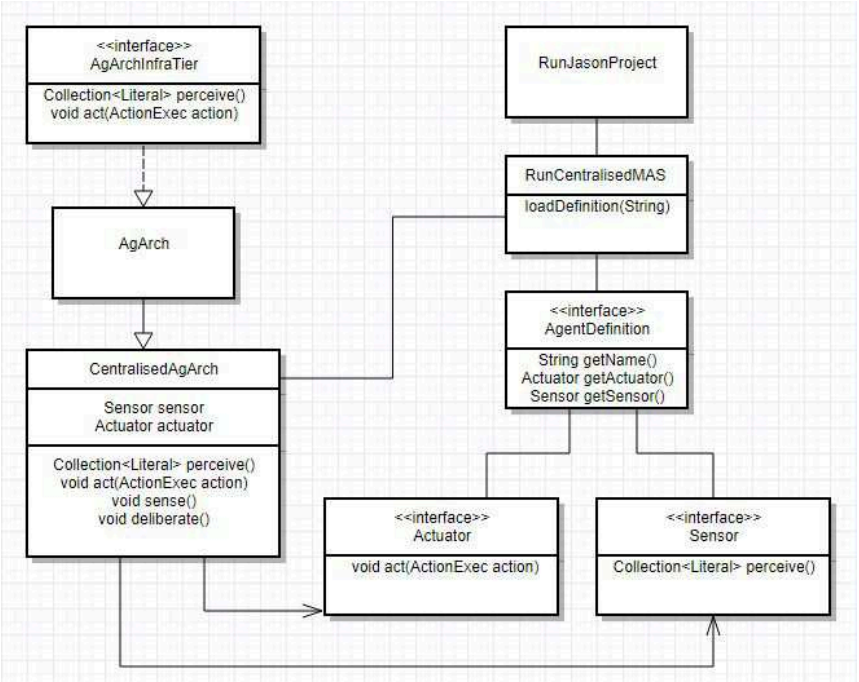
Na imagem podemos observar a primeira linha que apenas passa a responsabilidade para o sensor, e também a presença de outro trecho de código que visa permitir a retrocompatibilidade da arquitetura com projetos JASON que utilizam o ambiente do próprio JASON e utilizam os sensores e atuadores como forma de ponte entre os dois.

Da mesma forma aconteceu com o método *perceive(),* que também se encontrava vazio e foi modificado de forma a retornar o resultado do método *perceive()* do sensor, responsável por implementar a busca das percepções do agente. Sendo assim o método teve sua implementação da seguinte maneira:

```
@Override
public Collection<Literal> perceive() {
    return this.sensor.perceive();
}
```

Desta forma a nova arquitetura pode ser representada pelo diagrama:

Figura 7 - DIAGRAMA DE CLASSES SIMPLIFICADO DA ARQUITETURA IMPLEMENTADA



Feito isto foi realizado todo o processo de atualização do *.jar,* bem como um novo teste com um agente simples, que não realiza tarefa alguma de forma a garantir apenas que o código executa sem erros. O teste foi realizado com sucesso, porém se fez necessária a realização de um novo teste com agentes funcionais objetivando mostrar que o motor BDI continua funcionando após as modificações realizadas no fluxo de funcionamento dos agentes.

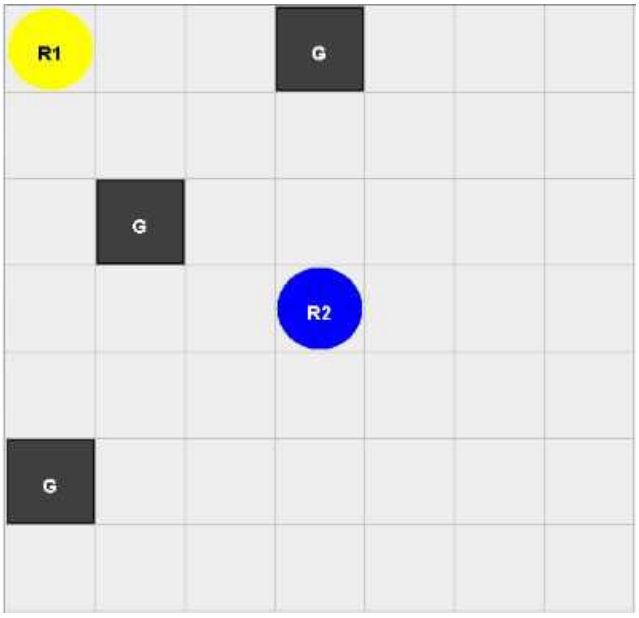
Neste ponto também foi considerou-se a possibilidade de utilizar o conceito de Artefatos na arquitetura para a descrição do ambiente, que consiste na presença de artefatos no ambiente que funcionam como ferramentas e fonte de informação. A utilização de tal conceito apesar de facilitar o desenvolvimento dos agentes restringiria a aplicabilidade da arquitetura proposta se tal conceito fosse agregado na parte interna da arquitetura. Desta forma caso o usuário considere interessante o uso de Artefatos existe a possibilidade de utilizá-lo no nível dos sensores e atuadores, implementado-os de forma que trabalhem utilizando o conceito de Artefatos.

4.3 TESTES PRELIMINARES

Para realizar um teste mais complexo optou-se por utilizar um projeto de teste disponibilizado pela própria arquitetura JASON chamado Cleaning Robots, ou robôs de limpeza. Este projeto se faz interessante para o teste pois possui dois agentes com tarefas

bem definidas, uma interface gráfica simples tornando fácil a verificação do correto funcionamento do motor BDI. Este projeto consiste em dois agentes R1 e R2, situados em um cenário de mundo extremamente simples: um tabuleiro semelhante ao do xadrez.

Figura 8 - INTERFACE GRÁFICA DO AMBIENTE DO PROJETO CLEANING ROBOTS



O objetivo dos agentes é fazer a “limpeza” do cenário, sendo gerado em algumas posições do tabuleiro um objeto “lixo”. Cada um dos agentes possuem papéis diferentes, sendo eles:

- R1: Deve caminhar de quadrado em quadrado do tabuleiro, buscando encontrar o lixo. Uma vez encontrado um lixo, o agente deve recolhê-lo e leva-lo até o agente R2. Após entregar o lixo, o agente R1 deve voltar para a posição em que estava antes de encontrá-lo e continuar buscando outros lixos;
- R2: Fica parado esperando o R1 entregar o lixo. Quando recebe o lixo, é responsável apenas por queimá-lo. (Observe que a ação “queimar” consiste em apenas “deletar” o objeto lixo do cenário);

Para ser possível realizar a execução do projeto *Cleaning Robots* com as modificações feitas na arquitetura foi necessário criar as classes de definições para os agentes, chamadas de *r1.java* e *r2.java*. Objetivando realizar um teste inicial, foi criado um atuador e um sensor simples, com implementações de métodos vazias, ou seja que não realizam nada. Executando o projeto foi possível constatar que ele inicia sem erros e como esperado o agente R1 não se mexe - já que o atuador não realiza papel algum. Na sequência foi realizada a implementação do sensor: já que o projeto ainda possui uma

classe responsável por implementar o ambiente (e por consequência possui métodos responsáveis por realizar a implementação de como é feita a captura das percepções do agente) foi optado por buscar a instância do ambiente (representado pela classe *MarsEnv.java*), e realizar a chamada do mesmo no sensor. Sendo assim o método *perceive()* ficou responsável apenas por realizar a chamada do método *getPercepts(String)* do ambiente. A implementação do atuador foi feita da mesma maneira, no método *act()* buscando a instância do ambiente e chamando o método *executeAction(Stirng,Structure).*

Observa-se que neste teste a responsabilidade de implementar a captura das percepções do agente e a implementação das ações não está realmente no atuador e no sensor, já que estes ficaram responsáveis apenas por repassar a responsabilidade para o ambiente. Ou seja, o sensor e atuador apenas fazem uma "ponte" entre a arquitetura do agente e o ambiente fornecido na implementação original do cenário. O modo como foi realizada a implementação do atuador e sensor neste teste não representa a forma como este trabalho propõe-se a realizar a implementação dos mesmos. Optou-se por realizá-la desta forma com a intenção de garantir que o motor BDI continua funcionando mesmo com a utilização destes componentes.

Realizada a implementação foi executado o teste e a execução ocorreu sem problemas: ambos os agentes executaram suas funções. Visando fazer uma comparação em termos de desempenho o Cleaning Robots também foi executado na versão do JASON sem as modificações deste trabalho e nenhuma diferença significativa foi constatada no tempo de execução.

Constatada esta necessidade optou-se por realizar mais um teste em um cenário mais próximo do cenário do projeto Awareness - onde será realizado o estudo de caso deste trabalho. O ambiente do projeto Awareness é realizado utilizando a tecnologia Unity que é um framework para desenvolvimento de cenários virtuais, mais especificamente para jogos. É multi-plataforma e permite a criação de cenários 2D ou 3D e conta com uma IDE própria chamada Unity Editor para o desenvolvimento de cenários em Unity. De maneira resumida o desenvolvimento de cenários é baseado em componentes: uma pessoa é um componente, uma parede é um componente e até a câmera do próprio jogador é um componente, cada um com suas propriedades específicas. Diferentes comportamentos podem ser adicionados aos componentes através de scripts.

O cenário do Awareness é de grande complexidade, mas objetivando realizar uma simulação inicial foi utilizado um projeto mais simples no Unity desenvolvido pelo bolsista do

projeto Awareness, acadêmico Paulo Lefol, mas que em paralelo possui os requisitos necessários para realizar este teste. Consiste em um cenário 3D onde o corpo do agente se encontra em um terreno plano, e a sua volta existem “fotos” de alguns personagens de filme. Cada um desses quadros possui como propriedade um nome.

Neste cenário o objetivo do agente consiste em encontrar uma foto em seu cenário. Como os objetos estão distribuídos em volta do agente, basta que ele rotacione sua visão para os lados até que encontre uma foto. Buscando simplificar a implementação, ficou estabelecido então que o agente tem como objetivo rotacionar a sua visão para a direita até que encontre uma foto.

Antes de dar início ao desenvolvimento do projeto JASON, que implementaria o agente deste cenários, um problema foi identificado: Uma vez que o Unity é um processo separado do JASON, como realizar a comunicação entre eles? Existe um agravante neste comunicação, que é o fato de o Unity ser desenvolvido em uma linguagem diferente (C#)[8] do JASON (Java).

Com o objetivo de elencar possíveis tecnologias e ferramentas a serem utilizadas como ponte entre os dois processos foi dado inicio á um estudo e dentre as possíveis tecnologias e a que demonstrou melhor adesão às necessidades específicas deste trabalho foi a utilização de *Socket*. O socket consiste basicamente em uma comunicação onde uma entidade atua como cliente e outra como servidor. Uma vez estabelecida uma conexão cliente-servidor ambos estão livres para trocar mensagens em uma espécie de conversa. Dentre as vantagens do socket pode-se destacar:

- Tecnologia altamente conhecida, com grande suporte em fóruns on-line;
- Suporte das principais linguagens de programação;
- Diversas APIs em diferentes linguagens para facilitar sua utilização;
- Permite que o servidor e o cliente estejam na mesma máquina, ou em máquinas diferentes;

Neste teste uma comunicação simples em socket foi implementada: o ambiente que roda em Unity irá atuar como o servidor socket e o JASON irá atuar como cliente. A comunicação é reduzida e não existe troca de dados complexos entre os participantes, apenas de números representando o tipo de requisição, e uma String representando a

[8] https://docs.microsoft.com/pt-br/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework

possível resposta. Sendo assim, o servidor socket foi projetado para atuar da seguinte maneira:

1. Aguarda uma conexão do cliente.
2. Verificar o tipo de mensagem na conexão, podendo a mensagem ser de dois tipos:
    a. Requisição de visão: no caso de uma requisição de visão o ambiente deve responder com o quê está o agente está vendo. Essa requisição é feita pelo sensor, para que ele possa gerar as percepções do agente a partir do que o agente “vê” no ambiente.
    b. Requisição de ação: no caso de uma requisição de ação, o ambiente deve receber o comando de ação e executá-la em seu framework. A única ação implementada foi a “rotação” do agente, ou seja, fazer com que a visão do agente rotacione levemente para o lado.
3. Encerra a conexão.

Neste ponto foi identificado que seria necessário a implementação em unity de 3 pontos:

- Servidor socket;
- Script para identificar o que está a frente de um “objeto” (o agente é representado como um “objeto” dentro do ambiente Unity);
- Script para rotacionar a visão do agente;

O Unity tem um bom suporte a inserção de novos comportamentos a seus objetos e fluxos já existentes. Para o servidor socket foi adicionado um script que roda uma *thread* em paralelo e é responsável por manter o servidor ativo. Para a identificação da visão do agente foi utilizado um script que é ativado sempre que um novo objeto entra no campo de visão do mesmo, então o script tem apenas de armazenar o nome do objeto como atributo, e então quando necessário o servidor socket requisita ao script este atributo. Para rotacionar o agente foi apenas necessário buscar a referência do objeto que representa o agente, e utilizar um método do próprio Unity responsável por modificar o objeto fisicamente.

Neste ponto todos os aspectos do servidor e da conexão estão implementados, sendo assim o próximo passo é a implementação do sensor, atuador e da definição *.asl* do agente. O sensor foi definido em uma classe chamada *UnitySensor* e a imeplementação do seu método *perceive()* consiste em:

1. Iniciar a conexão *socket*
2. Enviar uma requisição de visão ao servidor
3. Verificar se existe um objeto a frente do agente (resposta não pode ser vazia)
4. Caso exista um objeto, cria um objeto do tipo *Literal* que representa a crença de que existe um objeto a frente do agente e retorna-a como resultado do método. Esta crença é representada pela expressão *foundSomething(r1).*

Já o atuador foi chamado de *UnityActuator* e a implementação do seu método *act(ActionExec)* é bastante simples e consiste em:

1. Verifica se ação a ser executada é do tipo *turnRight* (rotacionar a visão do agente).
2. Caso afirmativo, inicia a conexão socket e envia a requisição de ação. Caso contrário não faz nada.

Congruente com a baixa complexidade do objetivo do agente, a definição do agente consiste em:

- Definição do objetivo inicial: o agente é iniciado com o objetivo *check.*
- A definição do plano a ser executada sempre que o agente possuir o objetivo check é: verificar se existe a crença *foundSomething(r1)*: caso afirmativo não faz nada, caso negativo executa ação *turnRight(r1)* e adiciona *check á lista de objetivos novamente.*

Feitas as implementações, foi dado início à execução do projeto que ocorreu sem apresentar erros, tanto do lado do cliente (JASON) como do lado do servidor (Unity). Porém um problema foi constatado: o agente executou a ação *turnRight* apenas uma vez, mesmo sem ter encontrado um objeto a frente do agente. Ou seja, o agente executou a ação uma vez e ficou congelado após a execução da mesma. Deu-se início então a uma extensiva busca do motivo da falha, que indica um problema no fluxo de operação do motor BDI. Objetivando facilitar a busca foi necessário passar a utilizar a classe *RunCentralisedMas* como ponto de partida para execução do projeto JASON, pois através dela é possível executar o framework diretamente do código, sem ser necessário a geração de um *.jar* e possibilitando assim que o código seja depurado através da ferramenta Eclipse, facilitando o processo de reprodução e estudo da falha. Após uma série de reproduções da falha foi identificado que o motor BDI atua de forma síncrona sobre as ações: ele dá início a execução de um ação, e não começa uma nova ação até que seja dado um sinal ao agente de que a execução terminou, informando o resultado. A responsabilidade de dar este “sinal” ao agente era do ambiente, e já que o mesmo foi completamente desacoplado da estrutura

do agente, ninguém enviava este sinal e o agente ficava aguardando-o por tempo indeterminado. E é exatamente por este motivo que este erro foi detectado apenas aqui, e não no primeiro teste preliminar: como no primeiro teste os atuadores e sensores funcionavam apenas como uma “ponte” entre o ambiente e o agente, o ambiente ainda desempenhava o papel de dar o sinal para o agente de que a ação terminou.

Sendo assim a solução para o problema foi implementada no método *act(ActionExec)* da arquitetura do agente e consiste simplesmente em realizar a chamada do método *actionExecuted(ActionExec)* - que é o método que era chamado pelo ambiente de forma a dar o “sinal” que a ação tinha sido executada. Após esta correção uma nova execução do teste foi realizada e o agente executou seu papel perfeitamente: do ponto de vista do ambiente o agente foi rotacionando sua visão para o lado até que encontrasse um objeto, e então terminou sua execução. Com este teste foi possível concluir que o motor BDI continua funcionando corretamente mesmo com as modificações no fluxo.

4.4 ESTUDO DE CASO NO PROJETO AWARENESS

Conforme descrito anteriormente este trabalho tem como sua motivação inicial o projeto Awareness e também tem como um de seus objetivos específicos a realização de um estudo de caso da arquitetura desenvolvida nele. O projeto Awaraness tem como objetivo fazer um estudo acerca do foco de pessoas enquanto estas fazem uso de dispositivos móveis em situações como o trânsito de automóveis. Para realizar este estudo faz-se uso de uma simulação de realidade virtual 3D desenvolvida na plataforma Unity. Esta simulação consiste de um cenário de uma cidade urbana, onde tem-se uma via de automóveis, uma faixa de pedestres, um semáforo para os pedestres e carros. A visão do humano fica localizado próximo a faixa de pedestres e sua função dentro da simulação é controlar o semáforo de forma que os pedestres (representados por gatos) não colidam com os carros, alterando o sinal para vermelho quando não é seguro atravessar e alterando para verde quando é seguro.

Figura 7 - SIMULAÇÃO DE REALIDADE VIRTUAL DO PROJETO AWARENESS

De forma a simular distrações reais, a pessoa que está participando da simulação opera um smartphone e distrações programadas são enviadas para o aparelho de forma a poder detectar o impacto que estas distrações tem no foco da pessoa, que é responsável em paralelo por controlar o semáforo da simulação. Observe que, já que o objetivo da simulação é fazer uma análise acerca do foco nos seres humanos para que se obtenha resultados com fidelidade, é necessário uma simulação com um suficiente nível de representatividade, sendo assim os componentes da simulação (motoristas, pedestres, etc) devem possuir este nível, e para este tipo de comportamento computacional considera-se adequada a utilização de agentes. Porém é neste ponto que encontra-se a problemática que este trabalho aborda: as arquiteturas atuais para o desenvolvimento de agentes funcionam com ambientes mais simples, onde o ambiente encontra-se e controlado pelo próprio framework do agente, o que não é o caso do Awareness. Então, para isto, utiliza-se a arquitetura que este trabalho desenvolveu, de forma que os agentes são desenvolvidos em Java na arquitetura JASON e são integrados à simulação em Unity através da estrutura de atuadores e sensores que utiliza uma conexão socket. Sendo assim optou-se por realizar a implementação de um agente que representa o pedestre (gato) na simulação, o comportamento do pedestre consiste basicamente em parar de caminhar quando encontra o semáforo fechado e voltar a caminhar quando o mesmo está aberto.

O desenvolvimento do teste se deu em duas partes, sendo a primeira delas o desenvolvimento no que diz respeito aos agentes, em Java. Para isso foi criado um projeto JASON chamado *awareness_lemming*. Neste projeto o primeiro passo foi criar a definição do agente chamado *lemming* no arquivo *lemming.asl.* Para realizar a definição utilizou-se a

lógica de eventos: eventos são disparados ao serem adicionadas novas crenças ao agente: quando a crença *redLight* é adicionada ao agente, o mesmo adiciona na lista de intenções a intenção *stopWalking*, e quando a crença *greenLight* é adicionado adiciona a intenção *startWalking*.

O proximo passo foi a criação da classe *lemming.java,* que implementa a interface *AgentDefinition,* de forma a informar o atuador e o sensor do agente, representados pelas classes *LemmingUnityActuator* e *LemmingUnitySensor*, respectivamente. Ambos se comunicam com o Unity através de uma comunicação Socket. A comunicação Socket utilizada foi a mesma desenvolvida para o teste realizado com Unity anteriormente. O atuador pode enviar duas mensagens diferentes de acordo com a ação a ser executada, que é determinada pelas intenções do agente:

- No caso de uma intenção *startWalking* o atuador envia uma mensagem com o conteúdo “1” de forma a comunicar para o ambiente que o gato deve começar a andar - é necessário apenas enviar um sinal para que ele comece a andar, já que a implementação do ato de “andar” de fato (movimentar-se, escolher a direção) é implementada pelo próprio projeto Unity.
- No caso de uma intenção *stopWalking* o atuador envia a mensagem com o conteúdo “2” que comunica que o gato deve parar de andar;

Já o sensor sempre envia a mensagem com o conteúdo “0” e aguarda a resposta do ambiente. O conteúdo “0” indica ao Unity que quer-se saber se o semáforo está com a cor vermelha ligada. Desta forma o sensor verifica a cor do semáforo e traduz isso para uma crença: se a resposta for positiva (luz vermelha ativa) ele retorna a crença *redLight* e caso seja negativa (luz verde ativa) retorna a crença *greenLight.*

Desenvolvido o projeto JASON o próximo passo foi a implementação do lado Unity, um servidor socket que deve responder as mensagens que possuem quatro possíveis conteúdos:

- Requisição com conteúdo “0”: No caso desta requisição que busca saber se o semáforo está com a cor vermelha ativa, foi utilizado o componente *PuffinCrossing* que representa o semáforo, nele existe o método *isRed().* Desta forma basta realizar a chamada deste método e retornar o resultado.
- Requisição com conteúdo “1”: Já nesta requisição deve-se fazer com que o gato comece a andar. A classe *MoveToWayPointsLemming* possui o método *move(),* que é responsável por implementar de fato a movimentação do gato. Este método foi

modificado de forma a adicionar uma verificação em uma variável booleana *isMoving*, de forma que se esta variável seja setada para falso o gato não realiza a movimentação, permanecendo parado. Bastou então adicionar nesta classe métodos que permitem alterar para verdadeiro ou falso a variável booleana. Sendo assim, quando o servidor socket recebe esta requisição, ele tem apenas de fazer a chamada do método que seta a variável para verdadeiro.

- Requisição com conteúdo “2”: Esta requisição deve fazer com o que o gato pare de andar. Utiliza exatamente a mesma estrutura da requisição anterior, porém realiza a chamada do método que seta a variável para falso.
- Requisição com conteúdo “3”: Esta requisição não é utilizada pelo agente, e foi inserida para tornar possível a realização do teste: ela é responsável por alterar a cor atual do semáforo, alterando para verde caso esteja vermelha e vice-versa. Para realizar isto foi necessário apenas chamar o método *isRed()* do componente *PuffinCrossing* verificando se o sinal está com a luz vermelha acesa, e caso esteja chamada o método *changeToGreen()* e caso contrário chamada o método *changeToRed()*.

Para enviar uma mensagem com o conteúdo “3” foi desenvolvido uma classe java chamada *UnitySemaforoController,* que possui apenas um método *main* responsável apenas por enviar a mensagem através da mesma conexão socket utilizada pelo atuador e pelo sensor. Sendo assim esta classe pode ser executada a qualquer momento.

Feito o desenvolvimento de ambos os lados do teste, o mesmo estava pronto para ser executado. Para realizar o teste primeiro foi iniciada a simulação, e então foi iniciado o motor BDI do projeto JASON. Inicialmente o gato começou a caminhar sem problemas, e caminhava livremente pela faixa de pedestres, independente da presença de carros - já que o semáforo é iniciado sempre com a luz verde. No momento em que foi executada a classe *UnitySemaforoController,* mudando a luz para vermelho, o seguinte ocorreu:

- A crença redLight foi adicionada a lista de crenças do agente;
- O agente enviou uma requisição ao Unity para que o gato pare de andar;
- O gato parou de andar.

Ao executar a classe UnitySemaforoController novamente mudando a luz para verde, o mesmo processo descrito anteriormente ocorreu, porém com a crença greenLight e uma requisição para que o gato volte a andar e como esperado o gato voltou a andar. Sendo assim conclui-se que o teste foi um sucesso e que o agente comportou-se exatamente

como esperado. Através deste teste foi possível identificar que o motor BDI continua a funcionar corretamente mesmo em um contexto onde o framework do ambiente está separado do framework do agente, concluindo assim o último objetivo específico deste trabalho que consiste em realizar um estudo de caso acerca da implementação realizada, modelando agentes para representar os pedestres na simulação imersiva do projeto Awareness.

5 CONSIDERAÇÕES FINAIS

Este trabalho inicialmente destacou no capítulo 1 uma problemática presente nas arquiteturas de agentes atualmente disponíveis, buscou embasar-se na literatura sobre agentes e suas possíveis implementações no capítulo 2. Através dos estudos de trabalhos correlatos no capítulo 3 foi possível selecionar qual seria a melhor abordagem e melhor arquitetura de agentes a serem utilizadas para o desenvolvimento da arquitetura a qual este trabalho se propõe a realizar. No capítulo 4 foi implementada a arquitetura, realizado uma série de testes e correções sobre a mesma e ao final foi realizado um estudo de caso.

Considerando-se os testes realizados na arquitetura pode-se concluir que a arquitetura implementada cumpre o objetivo a qual este trabalho se propõe realizar, que é tornar factível o desenvolvimento de agentes cujo ambiente está fora do framework do próprio agente. Isto é obtido através dos componentes atuador e sensor, pois ocorre uma correta divisão de responsabilidade, fazendo com que o agente seja responsável apenas por implementar os componentes do BDI, com que o ambiente seja responsável apenas por implementar o que diz respeito a si mesmo, enquanto o atuador e sensor ficam responsáveis por implementar a forma de interação com o ambiente e a análise dos dados recebidos do ambiente de forma a criar as percepções.

Por consequência da correta divisão de responsabilidade tem-se como resultado uma arquitetura que dá suporte ao cenário do projeto Awareness, e também aos mais diversos cenários, expandindo de forma considerável a aplicabilidade do JASON. Além disso a arquitetura também promove boas práticas de desenvolvimento de software no que diz respeito à programação orientada a objetos, respeitando o conceito de encapsulamento, aumentando a manutenibilidade e testabilidade dos projetos desenvolvidos nela. Conclui-se também que já que a forma de comunicação com o ambiente, encontra-se totalmente isolada no atuador e sensor, é possível mudar completamente o ambiente onde um agente atua, modificando apenas o atuador e sensor sem ter de modificar nada na implementação do agente.

Uma limitação identificada na arquitetura é no caso de uma possível demora na resposta de uma requisição ao ambiente, quando se está por exemplo buscando informações para construir as crenças do agente. Neste caso já que o motor BDI precisa das crenças do agente para executar o ciclo de pensamento do agente, a chamada deste método é síncrona e o ciclo de pensamento terá de esperar até que o ambiente responda a requisição.

Notou-se também que apesar deste trabalho propor uma arquitetura para atuação de agentes em ambientes virtuais, a arquitetura desenvolvida é flexível ao ponto de poder ser aplicada também para a atuação de agentes em ambientes reais, em casos como por exemplo onde temos um agente virtual, controlando um objeto real (como um robô).

5.1 TRABALHOS FUTUROS

Uma primeira sugestão de trabalho futuro é abordar a problemática descrita na conclusão: o que fazer no caso de demora na resposta do ambiente na busca pelas percepções do agente? Em casos onde o sensor do agente tem de buscar algum dado no ambiente, pode existir alguma demora no tempo de resposta, e seria interessante modificar o motor BDI de forma que o agente pudesse vir a continuar o seu ciclo de raciocínio enquanto aguarda a resposta.

Uma segunda sugestão é para casos onde o agente tem de criar suas percepções através da observação do ambiente como um todo, ou seja, em vez de perguntar dados específicos ao ambiente como realizado no estudo de caso, o agente iria apenas pedir o que está em seu campo de visão e realizar uma interpretação dos dados recebidos de forma a construir as suas percepções. Isto permite que seja desenvolvido agentes de comportamento enriquecido, já que as percepções do agente sobre o ambiente são muito mais abrangentes. Desta forma seria de grande valor ter um novo componente acoplado ao sensor do agente, de forma que o sensor seja apenas responsável por implementar a comunicação com o ambiente, e passar os dados “brutos” recebidos para tal componente e que ele realize a tradução destes dados para crenças do agente.

6 REFERÊNCIAS

LUGER, George F. 2014. Inteligência artificial. 6. ed. São Paulo: Pearson Education do Brasil, c2014. xvii, 614 p. ISBN 9788581435503.

G1. Uber lança serviço de carros sem motorista nos Estados Unidos. Disponível em: <http://g1.globo.com/tecnologia/noticia/2016/09/uber-lanca-servico-de-carros-sem-motorista-nos-estados-unidos.html>. Acesso em: 21 nov. 2016.

WOOLDRIDGE, Michael. Intelligent Agents: The Key Concepts. Department of Computer Science. University of Liverpool. 2002.

VRS. CAVE Fully Immersive Virtual Reality. Disponível em: <http://www.vrs.org.uk/virtual-reality-environments/cave.html>. Acesso em: 21 nov. 2016.

WOOLDRIDGE, Michael; JENNINGS, Nicholas. Agent Theories, Architectures, and Languages: A Survey. 1995.

SHENDARKAR, Ameya; VASUDEVAN, Karthik. Crowd simulation for emergency response using BDI agents based on immersive virtual reality

BONSON, Jéssica. Aplicação de agentes & artefatos para o desenvolvimento de uma ferramenta de autoria de objetos de aprendizagem. 2012.

PERIN, Phellipe. Integração de aprendizado em agentes bdi: arquitetura para processamento de percepções de embodied agents, 2017

RUSSEL, Stuart; NORVIG, Peter. Inteligência artificial: tradução da segunda edição. [S.l.]: Elsevier: Campus, 2004

PASSARELLA, Ricardo. Desenvolvimento de uma abordagem para cooperação em sistemas multiagentes. 2016.

POKAHR, Alexander, BRAUBACH, Lars, LAMERSDORF, Winfried, JADEX: A BDI Reasoning Engine, Springer US, University of Hamburg, 2005

KRISTIANSEN, Raymond. Subsumption architecture applied to flight control using composite rotations. 2016

GAUDOU, Benoit. BDI agents in social simulations: a survey. 2016.

DASTANI, Mehdi. 2APL: A Practical Agent Programming Language, 2007

FISHER, M. et al. Computational logics and agents: A roadmap of current technologies and future trend. 1993.

BORDINI, R. H.; HUBNER, J. F. Jason: A java-based interpreter for an extended version of agentspeak. 2007.

RUSSEL, Stuart; NORVIG, Peter. Inteligência artificial: tradução da segunda edição. [S.l.]: Elsevier: Campus, 2004

APÊNDICE A - Código

O código produzido por este trabalho consiste em uma série de modificações no código já existente da arquitetura de agentes JASON. Levando em conta o fato de que o código de tal arquitetura é de grande tamanho, neste apêndice estão presentes apenas as classes modificadas. O código completo foi disponibilizado através de um repositório público que pode ser acessado pelo endereço digital: https://github.com/hprandi/jason

Arquivo jason/infra/centralised/CentralisedAgArch.java

```
public class CentralisedAgArch extends AgArch implements Runnable {

	private CentralisedExecutionControl infraControl = null;
	private BaseCentralisedMAS masRunner = BaseCentralisedMAS.getRunner();

	private String agName = "";
	private volatile boolean running = true;
	private Queue<Message> mbox = new ConcurrentLinkedQueue<>();
	protected Logger logger = Logger.getLogger(CentralisedAgArch.class.getName());

	private Sensor sensor;
	private Actuator actuator;

	public CentralisedAgArch(Sensor sensor, Actuator actuator) {
		this.sensor = sensor;
		this.actuator = actuator;
	}

	private static List<MsgListener> msgListeners = null;

	public static void addMsgListener(MsgListener l) {
		if (msgListeners == null) {
			msgListeners = new ArrayList<>();
		}
		msgListeners.add(l);
	}

	public static void removeMsgListener(MsgListener l) {
		msgListeners.remove(l);
	}
```

```
	/**
	 * Creates the user agent architecture, default architecture is
	 * jason.architecture.AgArch. The arch will create the agent that creates
	 * the TS.
	 */
	public void createArchs(List<String> agArchClasses, String agClass, ClassParameters bbPars, String asSrc, Settings stts, BaseCentralisedMAS masRunner) throws JasonException {
		try {
			this.masRunner = masRunner;
			Agent.create(this, agClass, bbPars, asSrc, stts);
			this.insertAgArch(this);

			this.createCustomArchs(agArchClasses);

			// mind inspector arch
			if (stts.getUserParameter(Settings.MIND_INSPECTOR) != null) {
				this.insertAgArch((AgArch) Class.forName(Config.get().getMindInspectorArchClassName()).newInstance());
				this.getFirstAgArch().init();
			}

			this.setLogger();
		} catch (Exception e) {
			this.running = false;
			throw new JasonException("as2j: error creating the agent class! - " + e.getMessage(), e);
		}
	}

	/** init the agent architecture based on another agent */
	public void createArchs(List<String> agArchClasses, Agent ag, BaseCentralisedMAS masRunner) throws JasonException {
		try {
			this.masRunner = masRunner;
			this.setTS(ag.clone(this).getTS());
			this.insertAgArch(this);

			this.createCustomArchs(agArchClasses);
```

```
                this.setLogger();
            } catch (Exception e) {
                this.running = false;
                throw new JasonException("as2j: error creating the agent class! - ", e);
            }
        }

        public void stopAg() {
            this.running = false;
            this.wake(); // so that it leaves the run loop
            if (this.myThread != null) {
                this.myThread.interrupt();
            }
            this.getTS().getAg().stopAg();
            this.getUserAgArch().stop(); // stops all archs
        }

        public void setLogger() {
            this.logger = Logger.getLogger(CentralisedAgArch.class.getName() + "." +
this.getAgName());
            if (this.getTS().getSettings().verbose() >= 0) {
                this.logger.setLevel(this.getTS().getSettings().logLevel());
            }
        }

        public Logger getLogger() {
            return this.logger;
        }

        public void setAgName(String name) throws JasonException {
            if (name.equals("self")) {
                throw new JasonException("an agent cannot be named 'self'!");
            }
            if (name.equals("percept")) {
                throw new JasonException("an agent cannot be named 'percept'!");
            }
            this.agName = name;
```

```
}

@Override
public String getAgName() {
	return this.agName;
}

public AgArch getUserAgArch() {
	return this.getFirstAgArch();
}

public void setEnvInfraTier(CentralisedEnvironment env) {
}

public CentralisedEnvironment getEnvInfraTier() {
	return null;
}

public void setControlInfraTier(CentralisedExecutionControl pControl) {
	this.infraControl = pControl;
}

public CentralisedExecutionControl getControlInfraTier() {
	return this.infraControl;
}

private Thread myThread = null;

public void setThread(Thread t) {
	this.myThread = t;
	this.myThread.setName(this.agName);
}

public void startThread() {
	this.myThread.start();
}

@Override
```

```
public boolean isRunning() {
	return this.running;
}

protected void sense() {
	TransitionSystem ts = this.getTS();

	int i = 0;
	do {
		ts.sense(); // must run at least once, so that perceive() is called
	} while (this.running && ++i < this.cyclesSense && !ts.canSleepSense());
}

// int sumDel = 0; int nbDel = 0;
protected void deliberate() {
	TransitionSystem ts = this.getTS();
	int i = 0;
	while (this.running && i++ < this.cyclesDeliberate && !ts.canSleepDeliberate()) {
		ts.deliberate();
	}
	// sumDel += i; nbDel++;
	// System.out.println("running del "+(sumDel/nbDel)+"/"+cyclesDeliberate);
}

// int sumAct = 0; int nbAct = 0;
protected void act() {
	TransitionSystem ts = this.getTS();

	int i = 0;
	int ca = this.cyclesAct;
	if (this.cyclesAct == 9999) {
		ca = ts.getC().getIntentions().size();
	}

	while (this.running && i++ < ca && !ts.canSleepAct()) {
		ts.act();
	}
	// sumAct += i; nbAct++;
```

```
                // System.out.println("running act "+(sumAct/nbAct)+"/"+ca);
        }

        protected void reasoningCycle() {
                this.sense();
                this.deliberate();
                this.act();
        }

        @Override
        public void run() {
                TransitionSystem ts = this.getTS();
                while (this.running) {
                        if (ts.getSettings().isSync()) {
                                this.waitSyncSignal();
                                this.reasoningCycle();
                                boolean isBreakPoint = false;
                                try {
                                        isBreakPoint =
ts.getC().getSelectedOption().getPlan().hasBreakpoint();
                                        if (this.logger.isLoggable(Level.FINE)) {
                                                this.logger.fine("Informing controller that I finished a
reasoning cycle " + this.getCycleNumber() + ". Breakpoint is " + isBreakPoint);
                                        }
                                } catch (NullPointerException e) {
                                        // no problem, there is no sel opt, no plan ....
                                }
                                this.informCycleFinished(isBreakPoint, this.getCycleNumber());
                        } else {
                                this.incCycleNumber();
                                this.reasoningCycle();
                                if (ts.canSleep()) {
                                        this.sleep();
                                }
                        }
                }
                this.logger.fine("I finished!");
        }
```

```
private Object sleepSync = new Object();
private int sleepTime = 50;

public static final int MAX_SLEEP = 1000;

public void sleep() {
	try {
		if (!this.getTS().getSettings().isSync()) {
			// logger.fine("Entering in sleep mode....");
			synchronized (this.sleepSync) {
				this.sleepSync.wait(this.sleepTime); // wait for messages
				if (this.sleepTime < MAX_SLEEP) {
					this.sleepTime += 100;
				}
			}
		}
	} catch (InterruptedException e) {
	} catch (Exception e) {
		this.logger.log(Level.WARNING, "Error in sleep.", e);
	}
}

@Override
public void wake() {
	synchronized (this.sleepSync) {
		this.sleepTime = 50;
		this.sleepSync.notifyAll(); // notify sleep method
	}
}

@Override
public void wakeUpSense() {
	this.wake();
}

@Override
public void wakeUpDeliberate() {
```

```
            this.wake();
        }

        @Override
        public void wakeUpAct() {
            this.wake();
        }

        // Default perception assumes Complete and Accurate sensing.
        @Override
        public Collection<Literal> perceive() {
            return this.sensor.perceive();
        }

        // this is used by the .send internal action in stdlib
        @Override
        public void sendMsg(Message m) throws ReceiverNotFoundException {
            // actually send the message
            if (m.getSender() == null) {
                m.setSender(this.getAgName());
            }

            CentralisedAgArch rec = this.masRunner.getAg(m.getReceiver());

            if (rec == null) {
                if (this.isRunning()) {
                    throw new ReceiverNotFoundException("Receiver '" + m.getReceiver() + "'
does not exist! Could not send " + m);
                } else {
                    return;
                }
            }
            rec.receiveMsg(m.clone()); // send a cloned message

            // notify listeners
            if (msgListeners != null) {
                for (MsgListener l : msgListeners) {
                    l.msgSent(m);
```

```
                }
        }
}

public void receiveMsg(Message m) {
        this.mbox.offer(m);
        this.wakeUpSense();
}

@Override
public void broadcast(jason.asSemantics.Message m) throws Exception {
        for (String agName : this.masRunner.getAgs().keySet()) {
                if (!agName.equals(this.getAgName())) {
                        m.setReceiver(agName);
                        this.sendMsg(m);
                }
        }
}

// Default procedure for checking messages, move message from local mbox to C.mbox
@Override
public void checkMail() {
        Circumstance C = this.getTS().getC();
        Message im = this.mbox.poll();
        while (im != null) {
                C.addMsg(im);
                if (this.logger.isLoggable(Level.FINE)) {
                        this.logger.fine("received message: " + im);
                }
                im = this.mbox.poll();
        }
}

public Collection<Message> getMBox() {
        return this.mbox;
}

/** called by the TS to ask the execution of an action in the environment */
```

```
@Override
public void act(ActionExec action) {
	this.actuator.act(action);

	if (this.masRunner.getEnvironmentInfraTier() == null) {
		action.setResult(true);
		this.actionExecuted(action);
	}
}

@Override
public boolean canSleep() {
	return this.mbox.isEmpty() && this.isRunning();
}

private Object syncMonitor = new Object();
private volatile boolean inWaitSyncMonitor = false;

/**
 * waits for a signal to continue the execution (used in synchronised
 * execution mode)
 */
private void waitSyncSignal() {
	try {
		synchronized (this.syncMonitor) {
			this.inWaitSyncMonitor = true;
			this.syncMonitor.wait();
			this.inWaitSyncMonitor = false;
		}
	} catch (InterruptedException e) {
	} catch (Exception e) {
		this.logger.log(Level.WARNING, "Error waiting sync (1)", e);
	}
}

/**
 * inform this agent that it can continue, if it is in sync mode and
 * waiting a signal
```

```
 */
public void receiveSyncSignal() {
	try {
		synchronized (this.syncMonitor) {
			while (!this.inWaitSyncMonitor && this.isRunning()) {
				// waits the agent to enter in waitSyncSignal
				this.syncMonitor.wait(50);
			}
			this.syncMonitor.notifyAll();
		}
	} catch (InterruptedException e) {
	} catch (Exception e) {
		this.logger.log(Level.WARNING, "Error waiting sync (2)", e);
	}
}

/**
 * Informs the infrastructure tier controller that the agent
 * has finished its reasoning cycle (used in sync mode).
 *
 * <p>
 * <i>breakpoint</i> is true in case the agent selected one plan
 * with the "breakpoint" annotation.
 */
public void informCycleFinished(boolean breakpoint, int cycle) {
	this.infraControl.receiveFinishedCycle(this.getAgName(), breakpoint, cycle);
}

@Override
public RuntimeServicesInfraTier getRuntimeServices() {
	return new CentralisedRuntimeServices(this.masRunner);
}

private RConf conf;

private int cycles = 1;

private int cyclesSense = 1;
```

```
private int cyclesDeliberate = 1;
private int cyclesAct = 5;

public void setConf(RConf conf) {
	this.conf = conf;
}

public RConf getConf() {
	return this.conf;
}

public int getCycles() {
	return this.cycles;
}

public void setCycles(int cycles) {
	this.cycles = cycles;
}

public int getCyclesSense() {
	return this.cyclesSense;
}

public void setCyclesSense(int cyclesSense) {
	this.cyclesSense = cyclesSense;
}

public int getCyclesDeliberate() {
	return this.cyclesDeliberate;
}

public void setCyclesDeliberate(int cyclesDeliberate) {
	this.cyclesDeliberate = cyclesDeliberate;
}

public int getCyclesAct() {
	return this.cyclesAct;
}
```

```
        public void setCyclesAct(int cyclesAct) {
                this.cyclesAct = cyclesAct;
        }

}
```

Classe Actutor

```
public interface Actuator {
   void act(ActionExec action);
}
```

Classe Sensor

```
public interface Sensor {
   Collection<Literal> perceive();
}
```

Arquivo src/main/java/jason/infra/virtual/CentralisedAgArchAsynchronous.java

```
public class CentralisedAgArchAsynchronous extends CentralisedAgArch implements Runnable {
   private SenseComponent senseComponent;
   private DeliberateComponent deliberateComponent;
   private ActComponent actComponent;

   private ExecutorService executorSense;
   private ExecutorService executorDeliberate;
   private ExecutorService executorAct;

   public Object objSense = new Object();
   public Object objDeliberate = new Object();
   public Object objAct = new Object();

   public CentralisedAgArchAsynchronous(Sensor sensor, Actuator actuator) {
      super(sensor, actuator);

      this.senseComponent = new SenseComponent(this);
      this.deliberateComponent = new DeliberateComponent(this);
      this.actComponent = new ActComponent(this);
```

```
}

@Override
public void wakeUpSense() {
    this.senseComponent.wakeUp();
}

@Override
public void wakeUpDeliberate() {
    this.deliberateComponent.wakeUp();
}

@Override
public void wakeUpAct() {
    this.actComponent.wakeUp();
}

public SenseComponent getSenseComponent() {
    return this.senseComponent;
}

public DeliberateComponent getDeliberateComponent() {
    return this.deliberateComponent;
}

public ActComponent getActComponent() {
    return this.actComponent;
}

public ExecutorService getExecutorSense() {
    return this.executorSense;
}

public ExecutorService getExecutorDeliberate() {
    return this.executorDeliberate;
}

public ExecutorService getExecutorAct() {
```

```
        return this.executorAct;
    }

    public void setExecutorAct(ExecutorService executorAct) {
        this.executorAct = executorAct;
    }

    public void setExecutorSense(ExecutorService executorSense) {
        this.executorSense = executorSense;
    }

    public void setExecutorDeliberate(ExecutorService executorDeliberate) {
        this.executorDeliberate = executorDeliberate;
    }

    public void setSenseComponent(SenseComponent senseComponent) {
        this.senseComponent = senseComponent;
    }

    public void addListenerToC(CircumstanceListener listener) {
        this.getTS().getC().addEventListener(listener);
    }

    @Override
    public void receiveMsg(Message m) {
        synchronized (this.objSense) {
            super.receiveMsg(m);
        }
    }

    /** called the the environment when the action was executed */
    @Override
    public void actionExecuted(ActionExec action) {
        synchronized (this.objAct) {
            super.actionExecuted(action);
        }
    }
}
```

Arquivo src/main/java/jason/infra/virtual/CentralisedAgArchForPool.java

```
public final class CentralisedAgArchForPool extends CentralisedAgArch {
    private volatile boolean isSleeping = false;
    private ExecutorService executor;

    public CentralisedAgArchForPool(Sensor sensor, Actuator actuator) {
        super(sensor, actuator);
    }

    public void setExecutor(ExecutorService e) {
        this.executor = e;
    }

    @Override
    public void sleep() {
        this.isSleeping = true;
        /*
         * Agent.getScheduler().schedule(new Runnable() {
         * public void run() {
         * wake();
         * }
         * }, MAX_SLEEP, TimeUnit.MILLISECONDS);
         */
    }

    @Override
    public void wake() {
        synchronized (this) {
            if (this.isSleeping) {
                this.isSleeping = false;
                this.executor.execute(this);
            }
        }
    }

    @Override
```

```
    public void run() {
        int number_cycles = this.getCycles();
        int i = 0;

        while (this.isRunning() && i++ < number_cycles) {
            this.reasoningCycle();
            synchronized (this) {
                if (this.getTS().canSleep()) {
                    this.sleep();
                    return;
                } else if (i == number_cycles) {
                    this.executor.execute(this);
                    return;
                }
            }
        }
    }
}
```

Arquivo src/main/java/jason/infra/virtual/CentralisedRuntimeServices.java

```
public class CentralisedRuntimeServices implements RuntimeServicesInfraTier {

    private static Logger logger = Logger.getLogger(CentralisedRuntimeServices.class.getName());

    protected BaseCentralisedMAS masRunner;

    public CentralisedRuntimeServices(BaseCentralisedMAS masRunner) {
        this.masRunner = masRunner;
    }

    protected CentralisedAgArch newAgInstance() {
        throw new RuntimeException("Operation not allowed.");
    }

    @Override
```

```
public String createAgent(String agName, String agSource, String agClass, List<String> archClasses, ClassParameters bbPars, Settings stts, Agent father) throws Exception {
    if (logger.isLoggable(Level.FINE)) {
        logger.fine("Creating centralised agent " + agName + " from source " + agSource + " (agClass=" + agClass + ", archClass=" + archClasses + ", settings=" + stts);
    }

    AgentParameters ap = new AgentParameters();
    ap.setAgClass(agClass);
    ap.addArchClass(archClasses);
    ap.setBB(bbPars);

    if (stts == null) {
      stts = new Settings();
    }

    String prefix = null;
    if (father != null && father.getASLSrc().startsWith(SourcePath.CRPrefix)) {
      prefix = SourcePath.CRPrefix + "/";
    }
    agSource = this.masRunner.getProject().getSourcePaths().fixPath(agSource, prefix);

    String nb = "";
    synchronized (logger) { // to avoid problems related to concurrent executions of .create_agent
      int n = 1;
      while (this.masRunner.getAg(agName + nb) != null) {
        nb = "_" + n++;
      }
      agName = agName + nb;

      CentralisedAgArch agArch = this.newAgInstance();
      agArch.setAgName(agName);
        agArch.createArchs(ap.getAgArchClasses(), ap.agClass.getClassName(), ap.getBBClass(), agSource, stts, this.masRunner);
      agArch.setEnvInfraTier(this.masRunner.getEnvironmentInfraTier());
      agArch.setControlInfraTier(this.masRunner.getControllerInfraTier());
      this.masRunner.addAg(agArch);
    }
```

```
    logger.fine("Agent " + agName + " created!");
    return agName;
}

@Override
public void startAgent(String agName) {
    // create the agent thread
    CentralisedAgArch agArch = this.masRunner.getAg(agName);
    Thread agThread = new Thread(agArch);
    agArch.setThread(agThread);
    agThread.start();
}

@Override
public AgArch clone(Agent source, List<String> archClasses, String agName) throws JasonException {
    // create a new infra arch
    CentralisedAgArch agArch = this.newAgInstance();
    agArch.setAgName(agName);
    agArch.setEnvInfraTier(this.masRunner.getEnvironmentInfraTier());
    agArch.setControlInfraTier(this.masRunner.getControllerInfraTier());
    this.masRunner.addAg(agArch);

    agArch.createArchs(archClasses, source, this.masRunner);

    this.startAgent(agName);
    return agArch.getUserAgArch();
}

@Override
public Set<String> getAgentsNames() {
    return this.masRunner.getAgs().keySet();
}

@Override
public int getAgentsQty() {
    return this.masRunner.getAgs().keySet().size();
}
```

```
    @Override
    public boolean killAgent(String agName, String byAg) {
        logger.fine("Killing centralised agent " + agName);
        CentralisedAgArch ag = this.masRunner.getAg(agName);
        if (ag != null && ag.getTS().getAg().killAcc(byAg)) {
            ag.stopAg();
            this.masRunner.delAg(agName);
            return true;
        }
        return false;
    }

    @Override
    public void stopMAS() throws Exception {
        this.masRunner.finish();
    }
}
```

Classe RunCentralisedMAS

```
public class RunCentralisedMAS extends BaseCentralisedMAS {

    private JButton btDebug;

    public RunCentralisedMAS() {
        super();
        runner = this;
    }

    @Override
    public boolean hasDebugControl() {
        return this.btDebug != null;
    }

    @Override
    public void enableDebugControl() {
        this.btDebug.setEnabled(true);
    }
```

```
    public static void main(String[] args) throws JasonException {
        RunCentralisedMAS r = new RunCentralisedMAS();
        runner = r;
        r.init(args);
        r.create();
        r.start();
        r.waitEnd();
        r.finish();
    }

    protected int init(String[] args) {
        String projectFileName = null;
        if (args.length < 1) {
            if (RunCentralisedMAS.class.getResource("/" + defaultProjectFileName) != null) {
                projectFileName = defaultProjectFileName;
                readFromJAR = true;
                    Config.get(false); // to void to call fix/store the configuration in this case everything is read from a
jar/jnlp file
            } else {
                System.out.println("Jason " + Config.get().getJasonVersion());
                System.err.println("You should inform the MAS project file.");
                 // JOptionPane.showMessageDialog(null,"You should inform the project file as a parameter.\n\nJason
version "+Config.get().getJasonVersion()+" library built on
                // "+Config.get().getJasonBuiltDate(),"Jason", JOptionPane.INFORMATION_MESSAGE);
                System.exit(0);
            }
        } else {
            projectFileName = args[0];
        }

        if (Config.get().getJasonJar() == null) {
            System.out.println("Jason is not configured, creating a default configuration");
            Config.get().fix();
        }

        this.setupLogger();
```

```
if (args.length >= 2) {
   if (args[1].equals("-debug")) {
      debug = true;
      Logger.getLogger("").setLevel(Level.FINE);
   }
}

// discover the handler
for (Handler h : Logger.getLogger("").getHandlers()) {
   // if there is a MASConsoleLogHandler, show it
   if (h.getClass().toString().equals(MASConsoleLogHandler.class.toString())) {
      MASConsoleGUI.get().getFrame().setVisible(true);
      MASConsoleGUI.get().setAsDefaultOut();
   }
}

int errorCode = 0;

try {
   if (projectFileName != null) {
      InputStream inProject;
      if (readFromJAR) {
         inProject = RunCentralisedMAS.class.getResource("/" + defaultProjectFileName).openStream();
         urlPrefix = SourcePath.CRPrefix + "/";
      } else {
         URL file;
         // test if the argument is an URL
         try {
            file = new URL(projectFileName);
            if (projectFileName.startsWith("jar")) {
               urlPrefix = projectFileName.substring(0, projectFileName.indexOf("!") + 1) + "/";
            }
         } catch (Exception e) {
            file = new URL("file:" + projectFileName);
         }
         inProject = file.openStream();
      }
      jason.mas2j.parser.mas2j parser = new jason.mas2j.parser.mas2j(inProject);
```

```
        project = parser.mas();
    } else {
        project = new MAS2JProject();
    }

    project.setupDefault();
    project.getSourcePaths().setUrlPrefix(urlPrefix);
    project.registerDirectives();
    // set the aslSrcPath in the include
    ((Include) DirectiveProcessor.getDirective("include")).setSourcePath(project.getSourcePaths());

    project.fixAgentsSrc();

    if (MASConsoleGUI.hasConsole()) {
        MASConsoleGUI.get().setTitle("MAS Console - " + project.getSocName());

        this.createButtons();
    }

    // runner.waitEnd();
    errorCode = 0;

} catch (FileNotFoundException e1) {
    logger.log(Level.SEVERE, "File " + projectFileName + " not found!");
    errorCode = 2;
} catch (ParseException e) {
    logger.log(Level.SEVERE, "Error parsing file " + projectFileName + "!", e);
    errorCode = 3;
} catch (Exception e) {
    logger.log(Level.SEVERE, "Error!?: ", e);
    errorCode = 4;
}

System.out.flush();
System.err.flush();

if (!MASConsoleGUI.hasConsole() && errorCode != 0) {
    System.exit(errorCode);
```

```
        }
        return errorCode;
    }

    /** create environment, agents, controller */
    protected void create() throws JasonException {
        this.createEnvironment();
        this.createAgs();
        this.createController();
    }

    /** start agents, .... */
    protected void start() {
        this.startAgs();
        this.startSyncMode();
    }

    @Override
    public synchronized void setupLogger() {
        if (readFromJAR) {
            try {
                    LogManager.getLogManager().readConfiguration(RunCentralisedMAS.class.getResource("/" +
logPropFile).openStream());
            } catch (Exception e) {
                Handler[] hs = Logger.getLogger("").getHandlers();
                for (int i = 0; i < hs.length; i++) {
                    Logger.getLogger("").removeHandler(hs[i]);
                }
                Handler h = new MASConsoleLogHandler();
                h.setFormatter(new MASConsoleLogFormatter());
                Logger.getLogger("").addHandler(h);
                Logger.getLogger("").setLevel(Level.INFO);
            }
        } else {
            // checks a local log configuration file
            if (new File(logPropFile).exists()) {
                try {
                    LogManager.getLogManager().readConfiguration(new FileInputStream(logPropFile));
```

```
            } catch (Exception e) {
                System.err.println("Error setting up logger:" + e);
            }
        } else {
            try {
                if (runner != null) {
                    LogManager.getLogManager().readConfiguration(this.getDefaultLogProperties());
                } else {

LogManager.getLogManager().readConfiguration(RunCentralisedMAS.class.getResource("/templates/"          +
logPropFile).openStream());
                }
            } catch (Exception e) {
                System.err.println("Error setting up logger:" + e);
                e.printStackTrace();
            }
        }
    }
  }

  protected InputStream getDefaultLogProperties() throws IOException {
    return RunCentralisedMAS.class.getResource("/templates/" + logPropFile).openStream();
  }

  protected void setupDefaultConsoleLogger() {
    Handler[] hs = Logger.getLogger("").getHandlers();
    for (int i = 0; i < hs.length; i++) {
      Logger.getLogger("").removeHandler(hs[i]);
    }
    Handler h = new ConsoleHandler();
    h.setFormatter(new MASConsoleLogFormatter());
    Logger.getLogger("").addHandler(h);
    Logger.getLogger("").setLevel(Level.INFO);
  }

  protected void createButtons() {
    this.createStopButton();
```

```
// add Button pause
this.createPauseButton();

// add Button debug
                                        this.btDebug = new JButton("Debug", new
ImageIcon(RunCentralisedMAS.class.getResource("/images/debug.gif")));
this.btDebug.addActionListener(new ActionListener() {
   @Override
   public void actionPerformed(ActionEvent evt) {
      RunCentralisedMAS.this.changeToDebugMode();
      RunCentralisedMAS.this.btDebug.setEnabled(false);
      if (runner.control != null) {
         try {
            runner.control.getUserControl().setRunningCycle(false);
         } catch (Exception e) {
         }
      }
   }
});
if (debug) {
   this.btDebug.setEnabled(false);
}
MASConsoleGUI.get().addButton(this.btDebug);

// add Button start
                              final JButton btStartAg = new JButton("New agent", new
ImageIcon(RunCentralisedMAS.class.getResource("/images/newAgent.gif")));
btStartAg.addActionListener(new ActionListener() {
   @Override
   public void actionPerformed(ActionEvent evt) {
         new StartNewAgentGUI(MASConsoleGUI.get().getFrame(), "Start a new agent to run in current
MAS", System.getProperty("user.dir"));
   }
});
MASConsoleGUI.get().addButton(btStartAg);

// add Button kill
                              final JButton btKillAg = new JButton("Kill agent", new
```

```
ImageIcon(RunCentralisedMAS.class.getResource("/images/killAgent.gif")));
        btKillAg.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent evt) {
                new KillAgentGUI(MASConsoleGUI.get().getFrame(), "Kill an agent of the current MAS");
            }
        });
        MASConsoleGUI.get().addButton(btKillAg);

        this.createNewReplAgButton();

        // add show sources button
                                        final JButton btShowSrc = new JButton("Sources", new
ImageIcon(RunCentralisedMAS.class.getResource("/images/list.gif")));
        btShowSrc.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent evt) {
                showProjectSources(project);
            }
        });
        MASConsoleGUI.get().addButton(btShowSrc);

    }

    protected void createPauseButton() {
                                        final JButton btPause = new JButton("Pause", new
ImageIcon(RunCentralisedMAS.class.getResource("/images/resume_co.gif")));
        btPause.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent evt) {
                if (MASConsoleGUI.get().isPause()) {
                    btPause.setText("Pause");
                    MASConsoleGUI.get().setPause(false);
                } else {
                    btPause.setText("Continue");
                    MASConsoleGUI.get().setPause(true);
                }
```

```
            }
        });
        MASConsoleGUI.get().addButton(btPause);
    }

    protected void createStopButton() {
        // add Button
                                                JButton btStop = new JButton("Stop", new
ImageIcon(RunCentralisedMAS.class.getResource("/images/suspend.gif")));
        btStop.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent evt) {
                MASConsoleGUI.get().setPause(false);
                runner.finish();
            }
        });
        MASConsoleGUI.get().addButton(btStop);
    }

    protected void createNewReplAgButton() {
        // Do nothing
    }

    protected void createEnvironment() throws JasonException {
                                                    if (project.getEnvClass() != null &&
!project.getEnvClass().getClassName().equals(jason.environment.Environment.class.getName())) {
            logger.fine("Creating environment " + project.getEnvClass());
            this.env = new CentralisedEnvironment(project.getEnvClass(), this);
        }
    }

    protected void createAgs() throws JasonException {

        RConf generalConf = RConf.fromString(project.getInfrastructure().getParameter(0));

        int nbAg = 0;
        Agent pag = null;
```

```
// create the agents
for (AgentParameters ap : project.getAgents()) {
   try {

      String agName = ap.name;

      for (int cAg = 0; cAg < ap.getNbInstances(); cAg++) {
         AgentDefinition definition = this.loadDefinition(agName);

         nbAg++;

         String numberedAg = agName;
         if (ap.getNbInstances() > 1) {
            numberedAg += cAg + 1;
            // cannot add zeros before, it causes many compatibility problems and breaks dynamic creation
            // numberedAg += String.format("%0"+String.valueOf(ap.qty).length()+"d", cAg + 1);
         }

         String nb = "";
         int n = 1;
         while (this.getAg(numberedAg + nb) != null) {
            nb = "_" + n++;
         }
         numberedAg += nb;

         logger.fine("Creating agent " + numberedAg + " (" + (cAg + 1) + "/" + ap.getNbInstances() + ")");
         CentralisedAgArch agArch;

         RConf agentConf;
         if (ap.getOption("rc") == null) {
            agentConf = generalConf;
         } else {
            agentConf = RConf.fromString(ap.getOption("rc"));
         }

         // Get the number of reasoning cycles or number of cycles for each stage
         int cycles = -1; // -1 means default value of the platform
         int cyclesSense = -1;
```

```
            int cyclesDeliberate = -1;
            int cyclesAct = -1;

            if (ap.getOption("cycles") != null) {
               cycles = Integer.valueOf(ap.getOption("cycles"));
            }
            if (ap.getOption("cycles_sense") != null) {
               cyclesSense = Integer.valueOf(ap.getOption("cycles_sense"));
            }
            if (ap.getOption("cycles_deliberate") != null) {
               cyclesDeliberate = Integer.valueOf(ap.getOption("cycles_deliberate"));
            }
            if (ap.getOption("cycles_act") != null) {
               cyclesAct = Integer.valueOf(ap.getOption("cycles_act"));
            }

            // Create agents according to the specific architecture
            if (agentConf == RConf.POOL_SYNCH) {
               agArch = new CentralisedAgArchForPool(definition.getSensor(), definition.getActuator());
            } else if (agentConf == RConf.POOL_SYNCH_SCHEDULED) {
                              agArch = new CentralisedAgArchSynchronousScheduled(definition.getSensor(),
definition.getActuator());
            } else if (agentConf == RConf.ASYNCH || agentConf == RConf.ASYNCH_SHARED_POOLS) {
                                    agArch = new CentralisedAgArchAsynchronous(definition.getSensor(),
definition.getActuator());
            } else {
               agArch = new CentralisedAgArch(definition.getSensor(), definition.getActuator());
            }

            agArch.setCycles(cycles);
            agArch.setCyclesSense(cyclesSense);
            agArch.setCyclesDeliberate(cyclesDeliberate);
            agArch.setCyclesAct(cyclesAct);

            agArch.setConf(agentConf);
            agArch.setAgName(numberedAg);
            agArch.setEnvInfraTier(this.env);
                if (generalConf != RConf.THREADED && cAg > 0 && ap.getAgArchClasses().isEmpty() &&
```

```
ap.getBBClass().equals(DefaultBeliefBase.class.getName())) {
                    // creation by cloning previous agent (which is faster -- no parsing, for instance)
                    agArch.createArchs(ap.getAgArchClasses(), pag, this);
                } else {
                    // normal creation
                        agArch.createArchs(ap.getAgArchClasses(), ap.agClass.getClassName(), ap.getBBClass(),
ap.asSource.toString(),
                            ap.getAsSetts(debug, project.getControlClass() != null), this);
                }
                this.addAg(agArch);

                pag = agArch.getTS().getAg();
              }
            } catch (Exception e) {
              logger.log(Level.SEVERE, "Error creating agent " + ap.name, e);
            }
          }

          if (generalConf != RConf.THREADED) {
            logger.info("Created " + nbAg + " agents.");
          }
        }

        private AgentDefinition loadDefinition(String agName) {
          try {
            Class<?> clazz = Class.forName(agName);
            Object object = clazz.newInstance();
            logger.log(Level.SEVERE, "Found agent " + agName + " definition!");

            if (!(object instanceof AgentDefinition)) {
              throw new InvalidActivityException();
            }
            AgentDefinition definition = (AgentDefinition) object;
            logger.log(Level.SEVERE, "Agent name is " + definition.getName());
            return definition;
          } catch (ClassNotFoundException e) {
              BaseCentralisedMAS.logger.log(Level.SEVERE, "Agent " + agName + " definition (" + agName +
".java) not found.");
```

```
        } catch (InvalidActivityException e) {
            logger.log(Level.SEVERE, "Agent " + agName + " definition must implement AgentDefinition");
        } catch (InstantiationException e) {
            BaseCentralisedMAS.logger.log(Level.SEVERE, e.getMessage());
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            BaseCentralisedMAS.logger.log(Level.SEVERE, e.getMessage());
            e.printStackTrace();
        }
        return null;
    }

    protected void createController() throws JasonException {
        ClassParameters controlClass = project.getControlClass();
        if (debug && controlClass == null) {
            controlClass = new ClassParameters(ExecutionControlGUI.class.getName());
        }
        if (controlClass != null) {
            logger.fine("Creating controller " + controlClass);
            this.control = new CentralisedExecutionControl(controlClass, this);
        }
    }

    protected void startAgs() {
        // run the agents
                                                    if (project.getInfrastructure().hasParameter("pool") ||
project.getInfrastructure().hasParameter("synch_scheduled") ||
project.getInfrastructure().hasParameter("asynch")
                || project.getInfrastructure().hasParameter("asynch_shared")) {
            this.createThreadPool();
        } else {
            this.createAgsThreads();
        }
    }

    /** creates one thread per agent */
    private void createAgsThreads() {
```

```
        int cyclesSense = 1;
        int cyclesDeliberate = 1;
        int cyclesAct = 5;

        if (project.getInfrastructure().hasParameters()) {
            if (project.getInfrastructure().getParametersArray().length > 2) {
                cyclesSense = Integer.parseInt(project.getInfrastructure().getParameter(1));
                cyclesDeliberate = Integer.parseInt(project.getInfrastructure().getParameter(2));
                cyclesAct = Integer.parseInt(project.getInfrastructure().getParameter(3));
            } else if (project.getInfrastructure().getParametersArray().length > 1) {
                                                    cyclesSense = cyclesDeliberate = cyclesAct =
Integer.parseInt(project.getInfrastructure().getParameter(1));
            }

             // logger.info("Creating a threaded agents." + "Cycles: " + cyclesSense + ", " + cyclesDeliberate + ", " +
cyclesAct);
        }

        for (CentralisedAgArch ag : this.ags.values()) {
            ag.setControlInfraTier(this.control);

            // if the agent hasn't override the values for cycles, use the platform values
            if (ag.getCyclesSense() == -1) {
                ag.setCyclesSense(cyclesSense);
            }
            if (ag.getCyclesDeliberate() == -1) {
                ag.setCyclesDeliberate(cyclesDeliberate);
            }
            if (ag.getCyclesAct() == -1) {
                ag.setCyclesAct(cyclesAct);
            }

            // create the agent thread
            Thread agThread = new Thread(ag);
            ag.setThread(agThread);
        }

                // logger.info("Creating threaded agents. Cycles: " + agTemp.getCyclesSense() + ", " +
```

```
agTemp.getCyclesDeliberate() + ", " + agTemp.getCyclesAct());

        for (CentralisedAgArch ag : this.ags.values()) {
            ag.startThread();
        }
    }

    private Set<CentralisedAgArch> sleepingAgs;

    private ExecutorService executor = null;

    private ExecutorService executorSense = null;
    private ExecutorService executorDeliberate = null;
    private ExecutorService executorAct = null;

    /** creates a pool of threads shared by all agents */
    private void createThreadPool() {
        this.sleepingAgs = Collections.synchronizedSet(new HashSet<CentralisedAgArch>());

        int maxthreads = 10;

        int maxthreadsSense = 1;
        int maxthreadsDeliberate = 1;
        int maxthreadsAct = 1;

        int cycles = 1;
        int cyclesSense = 1;
        int cyclesDeliberate = 1;
        int cyclesAct = 5;

        try {
            ClassParameters infra = project.getInfrastructure();
            RConf conf = RConf.fromString(infra.getParameter(0));
            if (conf == RConf.ASYNCH) {
                maxthreadsSense = Integer.parseInt(infra.getParameter(1));
                maxthreadsDeliberate = Integer.parseInt(infra.getParameter(2));
                maxthreadsAct = Integer.parseInt(infra.getParameter(3));
                if (infra.getParametersArray().length > 5) {
```

```
            cyclesSense = Integer.parseInt(infra.getParameter(4));
            cyclesDeliberate = Integer.parseInt(infra.getParameter(5));
            cyclesAct = Integer.parseInt(infra.getParameter(6));
        }
         logger.info("Creating agents with asynchronous reasoning cycle. Sense (" + maxthreadsSense + "),
Deliberate (" + maxthreadsDeliberate + "), Act (" + maxthreadsAct
            + "). Cycles: " + cyclesSense + ", " + cyclesDeliberate + ", " + cyclesAct);

        this.executorSense = Executors.newFixedThreadPool(maxthreadsSense);
        this.executorDeliberate = Executors.newFixedThreadPool(maxthreadsDeliberate);
        this.executorAct = Executors.newFixedThreadPool(maxthreadsAct);

      } else { // async shared and pool cases
        if (infra.getParametersArray().length > 1) {
          maxthreads = Integer.parseInt(infra.getParameter(1));
        }
        if (infra.getParametersArray().length > 4) {
          cyclesSense = Integer.parseInt(infra.getParameter(2));
          cyclesDeliberate = Integer.parseInt(infra.getParameter(3));
          cyclesAct = Integer.parseInt(infra.getParameter(4));
        }

        if (conf == RConf.ASYNCH_SHARED_POOLS) {
           logger.info("Creating agents with asynchronous reasoning cycle (shared). Sense, Deliberate, Act ("
+ maxthreads + "). Cycles: " + cyclesSense + ", "
              + cyclesDeliberate + ", " + cyclesAct);
                                  this.executorSense = this.executorDeliberate = this.executorAct =
Executors.newFixedThreadPool(maxthreads);

        } else { // pool c ases
          if (conf == RConf.POOL_SYNCH) {
            // redefine cycles
            if (infra.getParametersArray().length == 3) {
              cycles = Integer.parseInt(infra.getParameter(2));
            } else if (infra.getParametersArray().length == 6) {
              cycles = Integer.parseInt(infra.getParameter(5));
            } else {
              cycles = 5;
```

```
                }
            } else if (infra.getParametersArray().length == 3) {
                cyclesSense = cyclesDeliberate = cyclesAct = Integer.parseInt(infra.getParameter(2));
            }

            int poolSize = Math.min(maxthreads, this.ags.size());
             logger.info("Creating a thread pool with " + poolSize + " thread(s). Cycles: " + cyclesSense + ", " +
cyclesDeliberate + ", " + cyclesAct
                    + ". Reasoning Cycles: " + cycles);

            // create the pool
            this.executor = Executors.newFixedThreadPool(poolSize);
          }
        }
      } catch (Exception e) {
        logger.warning("Error getting the number of thread for the pool.");
      }

      // initially, add all agents in the tasks
      for (CentralisedAgArch ag : this.ags.values()) {
        if (ag.getCycles() == -1) {
          ag.setCycles(cycles);
        }
        if (ag.getCyclesSense() == -1) {
          ag.setCyclesSense(cyclesSense);
        }
        if (ag.getCyclesDeliberate() == -1) {
          ag.setCyclesDeliberate(cyclesDeliberate);
        }
        if (ag.getCyclesAct() == -1) {
          ag.setCyclesAct(cyclesAct);
        }

        if (this.executor != null) {
          if (ag instanceof CentralisedAgArchForPool) {
            ((CentralisedAgArchForPool) ag).setExecutor(this.executor);
          }
          this.executor.execute(ag);
```

```
        } else if (ag instanceof CentralisedAgArchAsynchronous) {
            CentralisedAgArchAsynchronous ag2 = (CentralisedAgArchAsynchronous) ag;

            ag2.addListenerToC(new CircumstanceListenerComponents(ag2));

            ag2.setExecutorAct(this.executorAct);
            this.executorAct.execute(ag2.getActComponent());

            ag2.setExecutorDeliberate(this.executorDeliberate);
            this.executorDeliberate.execute(ag2.getDeliberateComponent());

            ag2.setExecutorSense(this.executorSense);
            this.executorSense.execute(ag2.getSenseComponent());
        }
    }
}

/** an agent architecture for the infra based on thread pool */
protected final class CentralisedAgArchSynchronousScheduled extends CentralisedAgArch {
    public CentralisedAgArchSynchronousScheduled(Sensor sensor, Actuator actuator) {
        super(sensor, actuator);
    }

    private volatile boolean runWakeAfterTS = false;
    private int currentStep = 0;

    @Override
    public void sleep() {
        RunCentralisedMAS.this.sleepingAgs.add(this);
    }

    @Override
    public void wake() {
        if (RunCentralisedMAS.this.sleepingAgs.remove(this)) {
            RunCentralisedMAS.this.executor.execute(this);
        } else {
            this.runWakeAfterTS = true;
        }
```

```
}

@Override
public void sense() {
    int number_cycles = this.getCyclesSense();
    int i = 0;

    while (this.isRunning() && i < number_cycles) {
        this.runWakeAfterTS = false;
        this.getTS().sense();
        if (this.getTS().canSleepSense()) {
            if (this.runWakeAfterTS) {
                this.wake();
            }
            break;
        }
        i++;
    }

    if (this.isRunning()) {
        RunCentralisedMAS.this.executor.execute(this);
    }
}

@Override
public void deliberate() {
    super.deliberate();

    if (this.isRunning()) {
        RunCentralisedMAS.this.executor.execute(this);
    }
}

@Override
public void act() {
    super.act();

    if (this.isRunning()) {
```

```
            RunCentralisedMAS.this.executor.execute(this);
        }
    }

    @Override
    public void run() {
        switch (this.currentStep) {
        case 0:
            this.sense();
            this.currentStep = 1;
            break;
        case 1:
            this.deliberate();
            this.currentStep = 2;
            break;
        case 2:
            this.act();
            this.currentStep = 0;
            break;
        }
    }
}

protected void stopAgs() {
    // stop the agents
    for (CentralisedAgArch ag : this.ags.values()) {
        ag.stopAg();
    }
}

/** change the current running MAS to debug mode */
protected void changeToDebugMode() {
    try {
        if (this.control == null) {
                                                this.control = new CentralisedExecutionControl(new
ClassParameters(ExecutionControlGUI.class.getName()), this);
            for (CentralisedAgArch ag : this.ags.values()) {
                ag.setControlInfraTier(this.control);
```

```
                Settings stts = ag.getTS().getSettings();
                stts.setVerbose(2);
                stts.setSync(true);
                ag.getLogger().setLevel(Level.FINE);
                ag.getTS().getLogger().setLevel(Level.FINE);
                ag.getTS().getAg().getLogger().setLevel(Level.FINE);
            }
        }
    } catch (Exception e) {
        logger.log(Level.SEVERE, "Error entering in debug mode", e);
    }
}

protected void startSyncMode() {
    if (this.control != null) {
        // start the execution, if it is controlled
        try {
            Thread.sleep(500); // gives a time to agents enter in wait
            this.control.informAllAgsToPerformCycle(0);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

protected void waitEnd() {
    try {
        // wait a file called .stop___MAS to be created!
        File stop = new File(stopMASFileName);
        if (stop.exists()) {
            stop.delete();
        }
        while (!stop.exists()) {
            Thread.sleep(1500);
            /*
             * boolean allSleep = true;
             * for (CentralisedAgArch ag : ags.values()) {
             * //System.out.println(ag.getAgName()+"="+ag.canSleep());
```

```
            * allSleep = allSleep && ag.canSleep();
            * }
            * if (allSleep)
            * break;
            */
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

private Boolean runningFinish = false;

@Override
public void finish() {
    // avoid two threads running finish!
    synchronized (this.runningFinish) {
        if (this.runningFinish) {
            return;
        }
        this.runningFinish = true;
    }
    try {
        // creates a thread that guarantees system.exit(0) in 5 seconds
        // (the stop of agents can block)
        new Thread() {
            @Override
            public void run() {
                try {
                    sleep(5000);
                } catch (InterruptedException e) {
                }
                System.exit(0);
            }
        }.start();

        System.out.flush();
        System.err.flush();
```

```
        if (MASConsoleGUI.hasConsole()) { // should close first! (case where console is in pause)
            MASConsoleGUI.get().close();
        }

        if (this.control != null) {
            this.control.stop();
            this.control = null;
        }
        if (this.env != null) {
            this.env.stop();
            this.env = null;
        }

        this.stopAgs();

        runner = null;

        // remove the .stop___MAS file (note that GUI console.close(), above, creates this file)
        File stop = new File(stopMASFileName);
        if (stop.exists()) {
            stop.delete();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }

    System.exit(0);
}

/** show the sources of the project */
private static void showProjectSources(MAS2JProject project) {
    JFrame frame = new JFrame("Project " + project.getSocName() + " sources");
    JTabbedPane pane = new JTabbedPane();
    frame.getContentPane().add(pane);
    project.fixAgentsSrc();

    for (AgentParameters ap : project.getAgents()) {
```

```
            try {
                String tmpAsSrc = ap.asSource.toString();

                // read sources
                InputStream in = null;
                if (tmpAsSrc.startsWith(SourcePath.CRPrefix)) {
                                                                                                    in      =
RunCentralisedMAS.class.getResource(tmpAsSrc.substring(SourcePath.CRPrefix.length())).openStream();
                } else {
                    try {
                        in = new URL(tmpAsSrc).openStream();
                    } catch (MalformedURLException e) {
                        in = new FileInputStream(tmpAsSrc);
                    }
                }
                StringBuilder s = new StringBuilder();
                int c = in.read();
                while (c > 0) {
                    s.append((char) c);
                    c = in.read();
                }

                // show sources
                JTextArea ta = new JTextArea(40, 50);
                ta.setEditable(false);
                ta.setText(s.toString());
                ta.setCaretPosition(0);
                JScrollPane sp = new JScrollPane(ta);
                pane.add(ap.name, sp);
            } catch (Exception e) {
                logger.info("Error:" + e);
            }
        }
        frame.pack();
        frame.setVisible(true);
    }
}
```

Arquivo src/main/java/jason/infra/virtual/AgentDefinition.java

```
public interface AgentDefinition {

    String getName();

    Actuator getActuator();

    Sensor getSensor();

}
```

Arquivo src/test/java/jason/asunit/TestArch.java

```
public class TestArch extends CentralisedAgArch implements Runnable {

    private static int nameCount = 0;

    private Condition condition;
    private int cycle = 0;

    private List<Literal> actions = new ArrayList<>();

    StringBuilder output = new StringBuilder();

    public TestArch() {
        this("ASUnitTest" + nameCount++);
    }

    public TestArch(String agName) {
        super(new SimpleSensor(), new SimpleActuator());
        try {
            this.setAgName(agName);
            BaseCentralisedMAS.getRunner().addAg(this);
        } catch (JasonException e) {
            e.printStackTrace();
        }
    }
```

```
    public int getCycle() {
        return this.cycle;
    }

    public List<Literal> getActions() {
        return this.actions;
    }

    public void start(Condition c) {
        this.condition = c;
        this.cycle = 0;
        this.actions.clear();
        new Thread(this).start();
    }

    @Override
    public void run() {
        synchronized (this.condition) {
            while (this.condition.test(this)) {
                this.cycle++;
                this.getTS().reasoningCycle();
                if (this.getTS().canSleep()) {
                    try {
                        Thread.sleep(50);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
            this.condition.notifyAll();
        }
    }

    public void setEnv(Environment env) {
        try {
                                    CentralisedEnvironment infraEnv = new CentralisedEnvironment(null,
BaseCentralisedMAS.getRunner());
```

```
            infraEnv.setUserEnvironment(env);
            env.setEnvironmentInfraTier(infraEnv);
            this.setEnvInfraTier(infraEnv);
        } catch (JasonException e) {
            e.printStackTrace();
        }
    }

    @Override
    public Collection<Literal> perceive() {
        // System.out.println(super.perceive()+"*"+getEnvInfraTier());
        if (this.getEnvInfraTier() != null) {
            return super.perceive();
        } else {
            return null;
        }
    }

    @Override
    public void act(ActionExec action) { // , List<ActionExec> feedback) {
        this.actions.add(action.getActionTerm());
        if (this.getEnvInfraTier() != null) {
            super.act(action); // , feedback); //env.scheduleAction(getAgName(), action.getActionTerm(), action);
        } else {
            action.setResult(true);
            this.actionExecuted(action); // feedback.add(action);
        }
    }

    public void print(String s) {
        System.out.println(s);
        this.output.append(s + "\n");
    }

    public StringBuilder getOutput() {
        return this.output;
    }
```

```
    public void clearOutput() {
        this.output = new StringBuilder();
    }
}
```

Arquivo src/main/java/jason/environment/Environment.java

```
public class Environment {

    private static Logger logger = Logger.getLogger(Environment.class.getName());

    private List<Literal> percepts = Collections.synchronizedList(new ArrayList<Literal>());
    private Map<String, List<Literal>> agPercepts = new ConcurrentHashMap<>();

    private boolean isRunning = true;

    /** the infrastructure tier for environment (Centralised, Saci, ...) */
    private EnvironmentInfraTier environmentInfraTier = null;

    // set of agents that already received the last version of perception
    private Set<String> uptodateAgs = Collections.synchronizedSet(new HashSet<String>());

    protected ExecutorService executor; // the thread pool used to execute actions

    /** creates an environment class with n threads to execute actions required by the agents */
    public Environment(int n) {
        // creates a thread pool with n threads
        this.executor = Executors.newFixedThreadPool(n);

        // creates and executor with 1 core thread
        // where no more than 3 tasks will wait for a thread
        // The max number of thread is 1000 (so the 1001 task will be rejected)
        // Threads idle for 10 sec. will be removed from the pool
                                // executor= new ThreadPoolExecutor(1,1000,10,TimeUnit.SECONDS,new
ArrayBlockingQueue<Runnable>(3));
    }
```

```
/** creates an environment class with the default number of threads in the pool */
public Environment() {
    this(4);
}

/**
 * Called before the MAS execution with the args informed in
 * .mas2j project, the user environment could override it.
 */
public void init(String[] args) {
}

/**
 * Called just before the end of MAS execution, the user
 * environment could override it.
 */
public void stop() {
    this.isRunning = false;
    this.executor.shutdownNow();
}

public boolean isRunning() {
    return this.isRunning;
}

/**
 * Sets the infrastructure tier of the environment (saci, jade, centralised, ...)
 */
public void setEnvironmentInfraTier(EnvironmentInfraTier je) {
    this.environmentInfraTier = je;
}

public EnvironmentInfraTier getEnvironmentInfraTier() {
    return this.environmentInfraTier;
}

public Logger getLogger() {
```

```
    return logger;
}

/**
 * @deprecated use version with String... parameter
 */
@Deprecated
public void informAgsEnvironmentChanged(Collection<String> agents) {
    if (this.environmentInfraTier != null) {
        this.environmentInfraTier.informAgsEnvironmentChanged(agents);
    }
}

public void informAgsEnvironmentChanged(String... agents) {
    if (this.environmentInfraTier != null) {
        this.environmentInfraTier.informAgsEnvironmentChanged(agents);
    }
}

/**
 * Returns percepts for an agent. A full copy of both common
 * and agent's percepts lists is returned.
 *
 * It returns null if the agent's perception doesn't changed since
 * last call.
 *
 * This method is to be called by TS and should not be called
 * by other objects.
 */
public Collection<Literal> getPercepts(String agName) {

    // check whether this agent needs the current version of perception
    if (this.uptodateAgs.contains(agName)) {
        return null;
    }
    // add agName in the set of updated agents
    this.uptodateAgs.add(agName);
```

```
    int size = this.percepts.size();
    List<Literal> agl = this.agPercepts.get(agName);
    if (agl != null) {
        size += agl.size();
    }
    Collection<Literal> p = new ArrayList<>(size);

    if (!this.percepts.isEmpty()) { // has global perception?
        synchronized (this.percepts) {
            // make a local copy of the environment percepts
            // Note: a deep copy will be done by BB.add
            p.addAll(this.percepts);
        }
    }
    if (agl != null) { // add agent personal perception
        synchronized (agl) {
            p.addAll(agl);
        }
    }

    return p;
}

/**
 * Returns a copy of the perception for an agent.
 *
 * It is the same list returned by getPercepts, but
 * doesn't consider the last call of the method.
 */
public List<Literal> consultPercepts(String agName) {
    int size = this.percepts.size();
    List<Literal> agl = this.agPercepts.get(agName);
    if (agl != null) {
        size += agl.size();
    }
    List<Literal> p = new ArrayList<>(size);

    if (!this.percepts.isEmpty()) { // has global perception?
```

```
        synchronized (this.percepts) {
            // make a local copy of the environment percepts
            // Note: a deep copy will be done by BB.add
            p.addAll(this.percepts);
        }
    }
    if (agl != null) { // add agent personal perception
        synchronized (agl) {
            p.addAll(agl);
        }
    }
    return p;
}

/** Adds a perception for all agents */
public void addPercept(Literal... perceptions) {
    if (perceptions != null) {
        for (Literal per : perceptions) {
            if (!this.percepts.contains(per)) {
                this.percepts.add(per);
            }
        }
        this.uptodateAgs.clear();
    }
}

/** Removes a perception from the common perception list */
public boolean removePercept(Literal per) {
    if (per != null) {
        this.uptodateAgs.clear();
        return this.percepts.remove(per);
    }
    return false;
}

/**
 * Removes all percepts from the common perception list that unifies with <i>per</i>.
 *
```

```
     * Example: removePerceptsByUnif(Literal.parseLiteral("position(_)")) will remove
     * all percepts that unifies "position(_)".
     *
     * @return the number of removed percepts.
     */
    public int removePerceptsByUnif(Literal per) {
        int c = 0;
        if (!this.percepts.isEmpty()) { // has global perception?
            synchronized (this.percepts) {
                Iterator<Literal> i = this.percepts.iterator();
                while (i.hasNext()) {
                    Literal l = i.next();
                    if (new Unifier().unifies(l, per)) {
                        i.remove();
                        c++;
                    }
                }
            }
            if (c > 0) {
                this.uptodateAgs.clear();
            }
        }
        return c;
    }

    /** Clears the list of global percepts */
    public void clearPercepts() {
        if (!this.percepts.isEmpty()) {
            this.uptodateAgs.clear();
            this.percepts.clear();
        }
    }

    /** Returns true if the list of common percepts contains the perception <i>per</i>. */
    public boolean containsPercept(Literal per) {
        if (per != null) {
            return this.percepts.contains(per);
        }
```

```
    return false;
}

/** Adds a perception for a specific agent */
public void addPercept(String agName, Literal... per) {
    if (per != null && agName != null) {
        List<Literal> agl = this.agPercepts.get(agName);
        if (agl == null) {
            agl = Collections.synchronizedList(new ArrayList<Literal>());
            this.agPercepts.put(agName, agl);
        }
        for (Literal p : per) {
            if (!agl.contains(p)) {
                this.uptodateAgs.remove(agName);
                agl.add(p);
            }
        }
    }
}

/** Removes a perception for an agent */
public boolean removePercept(String agName, Literal per) {
    if (per != null && agName != null) {
        List<Literal> agl = this.agPercepts.get(agName);
        if (agl != null) {
            this.uptodateAgs.remove(agName);
            return agl.remove(per);
        }
    }
    return false;
}

/**
 * Removes from an agent perception all percepts that unifies with <i>per</i>.
 *
 * @return the number of removed percepts.
 */
public int removePerceptsByUnif(String agName, Literal per) {
```

```
    int c = 0;
    if (per != null && agName != null) {
        List<Literal> agl = this.agPercepts.get(agName);
        if (agl != null) {
            synchronized (agl) {
                Iterator<Literal> i = agl.iterator();
                while (i.hasNext()) {
                    Literal l = i.next();
                    if (new Unifier().unifies(l, per)) {
                        i.remove();
                        c++;
                    }
                }
            }
            if (c > 0) {
                this.uptodateAgs.remove(agName);
            }
        }
    }
    return c;
}

public boolean containsPercept(String agName, Literal per) {
    if (per != null && agName != null) {
        @SuppressWarnings("rawtypes")
        List agl = this.agPercepts.get(agName);
        if (agl != null) {
            return agl.contains(per);
        }
    }
    return false;
}

/** Clears the list of percepts of a specific agent */
public void clearPercepts(String agName) {
    if (agName != null) {
        List<Literal> agl = this.agPercepts.get(agName);
        if (agl != null) {
```

```
            this.uptodateAgs.remove(agName);
            agl.clear();
        }
    }
}

/** Clears all perception (from common list and individual perceptions) */
public void clearAllPercepts() {
    this.clearPercepts();
    for (String ag : this.agPercepts.keySet()) {
        this.clearPercepts(ag);
    }
}

/**
 * Called by the agent infrastructure to schedule an action to be
 * executed on the environment
 */
public void scheduleAction(final String agName, final Structure action, final Object infraData) {
    this.executor.execute(new Runnable() {
        @Override
        public void run() {
            try {
                boolean success = Environment.this.executeAction(agName, action);
                Environment.this.environmentInfraTier.actionExecuted(agName, action, success, infraData); // send
the result of the execution to the agent
            } catch (Exception ie) {
                if (!(ie instanceof InterruptedException)) {
                    logger.log(Level.WARNING, "act error!", ie);
                }
            }
        }
    });
}

/**
 * Executes an action on the environment. This method is probably overridden in the user environment class.
 */
```

```
    public boolean executeAction(String agName, Structure act) {
        try {
            Thread.sleep(200);
        } catch (Exception e) {
        }
        this.informAgsEnvironmentChanged();
        return true;
    }
}
```

Arquivo testes/Teste1-Quadros/src/asl/r1.asl

```
!check(slots).
+!check(slots) : not foundSomething(r1)
        <- turnRight(r1); !check(slots).
```

Arquivo testes/Teste1-Quadros/src/java/UnityActuactor.java

```
public class UnityActuactor implements Actuator {

        public static final String turnRight = "turnRight";

        protected Logger logger = Logger.getLogger(UnityActuactor.class.getName());

        public void act(ActionExec action) {
                String functor = action.getActionTerm().getFunctor();

                this.logger.severe("Agent is acting.");

                if (functor.equals(turnRight)) {
                        this.logger.severe("Agent executing 'turnRight' action.");
                        try {
                                UnityConnection.sendMessage("1"); // rotate
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                }
```

```
	}

}
```

Arquivo testes/Teste1-Quadros/src/java/UnitySensor.java

```
public class UnitySensor implements Sensor {

	public static final Literal foundSomethingR1 = Literal.parseLiteral("foundSomething(r1)");

	public Collection<Literal> perceive() {
		Collection<Literal> perceptions = new ArrayList<Literal>();

		String objectName = null;

		try {
			objectName = UnityConnection.sendMessage("0"); // get perceptions
		} catch (IOException e) {
			e.printStackTrace();
		}

		if (objectName != null && !objectName.equals("none")) {
			perceptions.add(foundSomethingR1);
			perceptions.add(Literal.parseLiteral("found_" + objectName));
		}

		return perceptions;
	}

}
```

Arquivo testes/Teste1-Quadros/src/java/UnityConnection.java

```
public class UnityConnection {

	public static String sendMessage(String sentence) throws IOException {
		Socket clientSocket = new Socket("127.0.0.1", 27000);
```

```
                DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream());
                BufferedReader inFromServer = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

                outToServer.writeBytes(sentence + "\n");
                System.out.println("Sent to server: " + sentence);

                String response = inFromServer.readLine();
                System.out.println("Received from server:" + response);

                clientSocket.close();
                return response;
        }

}
```

Arquivo testes/Teste1-Quadros/src/java/r1.java

```
public class r1 implements AgentDefinition {

        public String getName() {
                return "r1";
        }

        public Sensor getSensor() {
                return new UnitySensor();
        }

        public Actuator getActuator() {
                return new UnityActuactor();
        }

}
```

Arquivo testes/Teste1-Quadros/testproject.mas2j

```
MAS testproject {
```

```
        infrastructure: Centralised

        agents:
                r1 r1;

        aslSourcePath:
                "src/asl";
}
```

Arquivo testes/Teste2-CleaningRobots/src/java/MarsActuator.java

```
public class MarsActuator implements Actuator {

        private String agName;

        public MarsActuator(String agName) {
                this.agName = agName;
        }

        public void act(ActionExec action) {
                MarsEnv.getInstance().scheduleAction(this.agName, action.getActionTerm(), action);
        }

}
```

Arquivo testes/Teste2-CleaningRobots/src/java/MarsSensor.java

```
public class MarsSensor implements Sensor {

        private String agName;

        public MarsSensor(String agName) {
                this.agName = agName;
```

```
	}

	public Collection<Literal> perceive() {
		return MarsEnv.getInstance().consultPercepts(this.agName);
	}

}
```

Arquivo testes/Teste2-CleaningRobots/src/java/r1.java

```
public class r1 implements AgentDefinition {

	public String getName() {
		return "r1";
	}

	public Sensor getSensor() {
		return new MarsSensor(this.getName());
	}

	public Actuator getActuator() {
		return new MarsActuator(this.getName());
	}

}
```

Arquivo testes/Teste2-CleaningRobots/src/java/r2.java

```
public class r2 implements AgentDefinition {

	public String getName() {
		return "r2";
	}

	public Sensor getSensor() {
		return new MarsSensor(this.getName());
	}
```

```
	public Actuator getActuator() {
		return new MarsActuator(this.getName());
	}

}
```

Arquivo testes/Teste3-AwarenessLemming/awareness_lemming.mas2j

```
MAS awareness_lemming {

	infrastructure: Centralised

	agents:
		lemming lemming;

	aslSourcePath:
		"src/asl";
}
```

Arquivo testes/Teste3-AwarenessLemming/src/asl/lemming.asl

```
+redLight : true
  <- stopWalking(lemming).

+greenLight : true
	<- startWalking(lemming).
```

Arquivo testes/Teste3-AwarenessLemming/src/java/Actions.java

```
public enum Actions {

	stopWalking,
	startWalking

}
```

Arquivo testes/Teste3-AwarenessLemming/src/java/Believes.java

```
public enum Believes {

	redLight,
	greenLight

}
```

Arquivo testes/Teste3-AwarenessLemming/src/java/LemmingUnityActuactor.java

```
public class LemmingUnityActuactor implements Actuator {
	protected Logger logger = Logger.getLogger(LemmingUnityActuactor.class.getName());

	public void act(ActionExec actionExec) {
		Actions action = Actions.valueOf(actionExec.getActionTerm().getFunctor());

		this.logger.severe("acting");

		if (action == null) {
			this.logger.severe("Action " + actionExec.getActionTerm().getFunctor() + " not
found.");
			return;
		}

		this.logger.severe("Agent executing '" + action.toString() + "' action.");
		boolean actionExecuted = false;

		switch (action) {
		case startWalking:
			actionExecuted = this.sendUnityMessage("1");
			break;
		case stopWalking:
			actionExecuted = this.sendUnityMessage("2");
			break;
		}

		if (actionExecuted) {
```

```
                this.logger.severe("Action executed.");
        }

        return;
    }

    private boolean sendUnityMessage(String str) {
        try {
            UnityConnection.sendMessage(str);
            return true;
        } catch (IOException e) {
            this.logger.severe("Error sending Unity message:");
            this.logger.severe(e.toString());
            return false;
        }
    }

}
```

Arquivo testes/Teste3-AwarenessLemming/src/java/LemmingUnitySensor.java

```
public class LemmingUnitySensor implements Sensor {

    int rounds = 0;

    public Collection<Literal> perceive() {
        Collection<Literal> perceptions = new ArrayList<Literal>();

        String lightStatus = null;
        try {
            lightStatus = UnityConnection.sendMessage("0"); // get light status
        } catch (IOException e) {
            e.printStackTrace();
        }

        Literal literal;
        if (lightStatus.equals("green")) {
```

```
                literal = Literal.parseLiteral(Believes.greenLight.toString());
            } else {
                literal = Literal.parseLiteral(Believes.redLight.toString());
            }

            perceptions.add(literal);

            return perceptions;
        }

}
```

Arquivo testes/Teste3-AwarenessLemming/src/java/UnityConnection.java

```
public class UnityConnection {

        public static String sendMessage(String sentence) throws IOException {
            Socket clientSocket = new Socket("127.0.0.1", 27000);

            DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream());
            BufferedReader inFromServer = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

            outToServer.writeBytes(sentence + "\n");
            System.out.println("Sent to server: " + sentence);

            String response = inFromServer.readLine();
            System.out.println("Received from server:" + response);

            clientSocket.close();
            return response;
        }

}
```

Arquivo testes/Teste3-AwarenessLemming/src/java/UnitySemaforoController.java

```
public class UnitySemaforoController {
        public static void main(String[] args) {
```

```
                try {
                        String sendMessage = UnityConnection.sendMessage("3");
                        System.out.println(sendMessage);
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

Arquivo testes/Teste3-AwarenessLemming/src/java/lemming.java

```
public class lemming implements AgentDefinition {

        public String getName() {
                return "lemming";
        }

        public Sensor getSensor() {
                return new LemmingUnitySensor();
        }

        public Actuator getActuator() {
                return new LemmingUnityActuactor();
        }

}
```

APÊNDICE B - Artigo

UMA ARQUITETURA PARA A ATUAÇÃO DE AGENTES INTELIGENTES

Henrique Prandi

Universidade Federal de Santa Catarina (UFSC)
Departamento de Informática e Estatística
Campus Universitário – Florianópolis – Brasil

henriqueprandi@gmail.com

Abstract. *In the context of virtual reality immersive simulations, it is common to exist cases where exists the necessity to have high complexity elements in the simulation, a lot of times simulating a simplified version of a human being. Aiming to solve this necessity this work proposes the use of the artificial intelligence's concept of intelligent agent, that is characterized by a system that can decide on its own what they need to do in order to satisfy its design goals (WOOLDRIDGE, 2002). Taking into account the fact that the use of intelligent agents in virtual reality immersive simulations its uncommon, this paper proposes the implementation of a architecture that allows agents to actuate on virtual 3D environments, and realize a study case with implementation of the proposed architecture on the Awareness project.*

Resumo. *No contexto de simulações imersivas de realidade virtual, é comum casos onde existe a necessidade da presença de elementos de alta complexidade na simulação, muitas vezes simulando uma versão simplificada de um humano. Buscando atender a esta necessidade este trabalho propõe a utilização do conceito da inteligência artificial de agente inteligente que caracteriza-se por sistemas que podem decidir por si próprios o que eles precisam fazer para satisfazer seus objetivos pré-designados (WOOLDRIDGE, 2002). Levando em conta o fato de que a utilização de agentes que atuam em simulações imersivas de realidade virtual é pouco comum, este trabalho propõe a implementação de uma arquitetura para agentes que permita a captura de informações e tomada de ações sobre ambientes virtuais.*

1. Introdução

Dentro dos diversos ramos da inteligência artificial, no contexto de tomada de decisão, temos o conceito de agentes. Pode-se definir agentes como sistemas capazes de decidir por si próprios o que eles precisam fazer para satisfazer seus objetivos pré-designados (WOOLDRIDGE, 2002). Uma das teorias de agentes é o modelo BDI: belief-desire-intention (em tradução livre: crenças-desejos-intenções), este modelo parte da idéia de que estes 3 pontos: crenças, desejos e intenções fazem parte fundamental do processo de tomada de decisão humano. Nesta arquitetura cada um dos 3 pontos desempenha um papel importante: as crenças são uma representação aproximada do estado atual do ambiente - é através desta representação que o agente decide o que é possível realizar ou não em um determinado

momento. Baseada nas opções do que é possível realizar naquele momento, os desejos funcionam como motivador para as ações do agente. Uma vez determinadas as possíveis opções de ação através das crenças, escolhidas quais ações são interessantes para a função específica daquele agente através dos desejos, é então escolhido um conjunto de ações a serem tomadas para atingir os objetivos, ações estas que são representadas na forma de intenções.

Este trabalho tem como motivação inicial o projeto Awareness, que visa construir um modelo de atenção para usuários de dispositivos móveis enquanto trafegam em vias de automóveis com o objetivo de prever o nível de atenção de um usuário. Utilizando o modelo construído combinado com informações obtidas pelo próprio dispositivo móvel do usuário, torna-se possível alertar o usuário sobre situações possivelmente perigosas de forma congruente com o seu nível de atenção.

Para construir um modelo de atenção o projeto utiliza de um ambiente de imersão virtual CAVE, um sistema de simulação onde uma pessoa encontra-se em uma sala com diversos aparatos para criar uma total imersão como por exemplo imagens projetadas nas paredes ao seu redor [VRS, 2016]. No caso do projeto a simulação é a de um usuário utilizando um dispositivo móvel em uma via automotiva. Nesta simulação estão presentes os principais elementos que caracterizam-se como importantes para a construção do modelo tais como carros (conduzidos por humanos), pedestres, semáforos e distrações.

Visando a construção de uma simulação com maior fidelidade, encontra-se a necessidade de simular os motoristas e pedestres. Nota-se que o comportamento dos seres humanos é de alta complexidade, definido pelas mais diversas características como sentimentos, desejos, vontades e motivações. Sendo assim é necessário um paradigma de desenvolvimento que permita representar estes humanos. Levando em conta que o ambiente onde o agente encontra-se é de alta complexidade - cenário pouco comum no contexto de desenvolvimento de agentes - este trabalho propõe uma arquitetura para que agentes atuem em ambientes de complexidade, como por exemplo o ambiente CAVE anteriormente descrito.

2. Conceitos básicos

Este capítulo apresenta três conceitos básicos necessários para o correto entendimento deste trabalho. Sendo tais conceitos: a definição de agente na seção 2.1, a definição e exemplos de teoria de agentes na seção 2.2, e por fim na seção 2.3 a definição e exemplos de arquiteturas e linguagens de agentes.

2.1. Agente

(WOOLDRIDGE, 2002) descreve que um agente é um sistema computacional que está situado em algum ambiente e é capaz de realizar ações autônomas nesse ambiente visando cumprir seus objetivos de modelagem. Aprofundando-se na definição de agente de (WOOLDRIDGE; JENNINGS, 1995) que define agentes principalmente através de sua capacidade em tomar decisões, tendo como palavra chave autonomia: um agente deve ser capaz de tomar decisões alinhadas com seus objetivos sem a intervenção ou supervisão de humanos ou outros sistemas. Esta mesma definição sugere que agentes devem ser

implementados através de conceitos-chave do processo de tomada de decisão humano como conhecimentos, crenças, intenções e obrigações.

(BONSON,2012) cita como exemplo de agente: um robô que explora uma área desconhecida a procura de um objeto, com atuadores e sensores para interagir com o ambiente ou então um agente jogador de xadrez com atuadores para mover as peças e sensores para compreender as jogadas do adversário.

2.2. Teoria de agentes

Para um correto entendimento do contexto de agentes, faz-se necessário o entendimento da Teoria de Agentes. Visando defini-la (BONSON,2012) cita a definição de (WOOLDRIDGE; JENNINGS, 1995): teorias de agentes são essencialmente especificações, que buscam definir aspectos tais como o que um agente é, e que propriedades ele deveria ter, e como pode-se representar e raciocinar sobre estas propriedades. É através da ótica proposta pela teoria de agentes, que algumas abordagens foram elaboradas, sobre como deve-se construí-los.

(BONSON,2012) descreve que atualmente a abordagem mais utilizada é a BDI (Belief-Desire-Intention) que envolve dois processos principais: decidir quais metas deseja-se atingir (deliberação) e como serão atingidas (meios-fins). Noções mentais humanas e uma perspectiva social são utilizadas na modelagem dos agentes. Os 3 conceitos básicos que compõem um agente BDI são:

- Crenças (Belief): Conjunto de crenças que o agente possui sobre o ambiente. É realizando uma observação do ambiente o qual está inserido, que o agente pode adquirir uma série de percepções do mesmo e com base nelas cria seu conjunto de crenças.
- Desejos: Conjunto de estados do ambiente que o agente deseja "atingir".
- Intenções: Sequência de ações que o agente se compromete a executar para atingir seus objetivos. Uma vez comprometido com uma intenção, o agente dará início à execução sequencial das ações daquela intenção, onde cada uma das ações consiste em de algum modo atuar sobre o ambiente o qual o agente está inserido.

Outra abordagem amplamente estudada é a de agentes reativos. De acordo com (WOOLDRIDGE, 2002) a abordagem reativa mais conhecida é a de subsunção, desenvolvida por Rodney Brooks. Nesta abordagem o processo de tomada de decisão é realizado através de um conjunto de comportamentos, no qual cada comportamento pode ser considerado como uma função que continuamente mapeia as percepções de entrada em uma ação a ser executada. Outra caracteríistica importante da arquitetura de subsunção é que vários comportamentos podem ser ativados ao mesmo tempo. (BONSON,2012) cita algumas vantagens da abordagem reativa: simplicidade, economia, tratabilidade computacional, robustez e elegância. E também algumas desvantagens como o fato de os agentes se basearem em informação local, sendo difícil tomarem decisões levando em consideração informações não-locais.

2.3. Linguagens e arquiteturas de agentes

Uma linguagem de agente é um sistema que permite programar e executar agentes, e que pode utilizar princípios estabelecidos pelas teorias de agentes (WOOLDRIDGE; JENNINGS, 1995).

A implementação de agentes também se faz possível através da utilização de uma arquitetura de agente, ao invés da utilização de uma linguagem. (WOOLDRIDGE; JENNINGS, 1995) propõe que a distinção entre linguagens e arquiteturas de agentes é de certa forma artificial, já que algumas das arquiteturas também podem ser consideradas linguagens. Boa parte do interesse em linguagens de agentes é motivado pelo conceito de programação orientada a agentes. Este paradigma baseia-se na idéia de de programar agentes nos termos e noções descritos pela teoria de agentes para que seja possível construir agentes nos mesmos termos que usamos para descrever o comportamento humano: através de lógica convencional.

Dentre as linguagens e arquiteturas citadas por (BONSON,2012) destacam-se para este trabalho:

- 2APL: É uma linguagem de programação orientada a agentes, onde define-se agentes através de crenças, objetivos, ações, planos, eventos e regras. Tem como um de seus principais objetivos facilitar o desenvolvimento de sistemas multi-agente, sendo assim fornece um arcabouço de funcionalidades em sua linguagem para cumprir este objetivo.
- AgentSpeak: É uma linguagem de programação orientada a agentes de alto nível onde define-se agentes através de crenças, eventos, objetivos, ações, planos e intenções.
- JASON: É uma arquitetura de agentes, que fornece estrutura para implementação de agentes em variados contextos. Nesta arquitetura os agentes são definidos através de uma versão estendida da linguagem de agentes AgentSpeak. A arquitetura JASON inclui um interpretador para esta linguagem e fornece toda a estrutura de forma a facilitar o desenvolvimento de agentes.
- JaCaMo: É uma arquitetura de agentes para o desenvolvimento de sistemas multi-agente. O JaCaMo é uma combinação de três outras tecnologias: JASON, Cartago e Moise. Cartago é uma infra-estruturada para o desenvolvimento de ambientes para sistema multi-agente baseada no modelo Agentes & Artefatos. Moise é um modelo para construção de organizações em sistemas multi-agente.
- JADEX: É uma arquitetura para o desenvolvimento de agentes baseado em componentes. Os agentes são definidos através de classes Java e arquivos XML. JADEX fornece uma boa estrutura no que diz respeito a engenharia de software.
- TouringMachines: Consiste de três camadas produtoras de atividades (camada reativa, camada de planejamento, camada de modelagem) que utilizam diferentes níveis de abstração para gerar continuamente sugestões sobre qual ação o agente deve tomar, e um sistema de controle,que decide qual camada terá controle sobre as ações do agente.

3. Escolha da teoria de agentes

Levando em conta o fato de que este trabalho tem como objetivo realizar a implementação de uma arquitetura para agentes que permita a captura de informações e

tomada de ações sobre ambientes virtuais é necessário escolher em qual teoria de agentes os agentes desta arquitetura serão embasados.

Existem algumas teoria de agentes na literatura sobre como o desenvolvimento de agentes deve ocorrer. Uma das teorias estudadas neste trabalho é a teoria reativa de subsunção. Esta teoria consiste em descrever o agente através de diversos comportamentos que atuam em conjunto para atingir o objetivo do mesmo. Esta arquitetura tem como algumas de suas principais vantagens a simplicidade, isto se dá pois é possível decompor comportamentos complexos em um conjunto de comportamentos de menor complexidade. Todavia observa-se também como desvantagem a fraca adaptabilidade da estrutura, já que comportamentos complexos são descritos em uma série de comportamentos menores, isto torna difícil modificar comportamentos já existentes ou adicionar novos. Também foi possível observar através dos trabalhos correlatos estudados na seção 3 que este tipo de arquitetura é mais adequada para cenários onde o agente tem de ter um comportamento mais reativo como o de um controlador de veículo aéreo, como foi o caso estudado. Observa-se que este tipo de cenário é totalmente diferente do cenário abordado neste trabalho.

Outra teoria é a BDI que propõe como principais componentes teóricos a definição de agentes através de crenças, desejos e intenções. Esta teoria destaca-se claramente como a mais adequada para este trabalho devido aos seguintes pontos:

- Flexibilidade: BDI permite uma alta flexibilidade e facilidade de alteração dos agentes implementados, pois a sua definição é feita através dos componentes teóricos, ou seja, não é necessário nenhum código "fixo" na arquitetura do agente para definir os comportamentos do mesmo. Já que os comportamentos ficam encapsulados, torna-se muito fácil a adição, remoção ou alteração dos mesmos;
- Comportamento enriquecido: A teoria escolhida parte da idéia de que os três componentes (crenças, desejos e intenções) são parte fundamental do processo de decisão humano, e simulando os três pode-se ter um comportamento de alta complexidade, com certeza não tão complexo como o humano, mas com representatividade suficiente. Sendo assim, se a intenção do agente é simular por exemplo um humano, ou outra entidade de comportamento complexo, o BDI faz-se uma ótima escolha.
- Utilização em trabalhos similares: Levando em conta o objetivo deste trabalho de tornar factível a atuação de agentes em ambientes virtuais, pode-se induzir que em geral os agentes implementados através desta arquitetura provavelmente serão agentes que representam humanos, ou pelo menos entidades com complexidade similar. Sendo assim analisando trabalhos correlatos e comparando os contextos dos mesmos com o deste trabalho, fica evidente que neste tipo de ambiente a arquitetura BDI é adequada.

4. Escolha da linguagem/arquitetura de agentes

Uma vez escolhida a abordagem de agentes, abre-se um leque de escolhas das linguagens e arquitetura existentes que implementam essa abordagem. Sendo assim faz-se necessário a escolha dentre as linguagens/arquiteturas elencadas na seção 2 deste trabalho que implementam a abordagem BDI.

Fazendo um comparativo entre as linguagens e arquiteturas acima, foi possível chegar a conclusões sobre cada uma delas. A linguagem 2APL tem seu principal foco no desenvolvimento de sistemas multi-agente, o que não considera-se essencial para este

trabalho. A linguagem AgentSpeak se enquadra como uma boa opção, porém existe a arquitetura JASON, que só não implementa uma versão ainda mais poderosa da AgentSpeak como também oferece outras vantagens no que diz respeito ao desenvolvimento de agentes. A arquitetura JaCaMo oferece uma série de pontos positivos devido à sua combinação de tecnologias, porém as tecnologias Cartago e Moise são focadas para o desenvolvimento de sistemas multi-agente, que como já citado não é o foco deste trabalho. A arquitetura JADEX também classifica-se como uma boa opção já que fornece bom suporte para o desenvolvimento de agentes e implementa boas práticas de programação orientada a objetos.

Levando em conta estas considerações, faz-se necessário um comparativo entre as duas arquiteturas consideradas mais indicadas para este trabalho: JASON e JADEX. É através desta comparação que alguns pontos muito positivos foram identificados sobre a arquitetura JASON, tornando-a a mais indicada para o contexto deste trabalho:

- Agentes descritos em linguagem interpretada: Os agentes da arquitetura JASON são descritos em arquivos individuais chamados .ASL, nestes arquivos os agentes são descritos em termos de BDI através da linguagem AgentSpeak que possui sintaxe e semântica específica para o contexto de agentes. Facilitando assim o desenvolvimento, interpretação, modificação e a manutenção dos agentes.
- Grande abstração: Devido a arquitetura e modo como os agentes são implementados o desenvolvedor não tem de se preocupar com comportamentos específicos da estrutura da arquitetura e nem com comportamentos do motor BDI. O desenvolvedor tem apenas de se preocupar com a correta definição das crenças, desejos e intenções do agente.
- Utilização em trabalhos similares: Assim como a abordagem da arquitetura JASON (BDI) a própria arquitetura JASON é utilizada em trabalhos com o contexto similar ao deste, formando assim um bom indicativo da congruẽncia da arquitetura com os objetivos deste trabalho.

4. Implementação da arquitetura para atuação de agentes em ambientes virtuais

Neste capítulo é apresentado como ocorreu o estudo da arquitetura escolhida e as conclusões sobre a mesma na seção 4.1 e as modificações realizadas na mesma na seção 4.2.

4.1. Estudo inicial sobre a arquitetura JASON

Alinhado com o objetivo deste trabalho deu-se início a implementação da arquitetura proposta. Inicialmente foi necessário uma pesquisa sobre a arquitetura JASON, buscando compreender como ela funciona, como é realizada execução de uma simulação e também foi realizado um estudo inicial sobre o código fonte.

A partir do estudo sobre o código fonte realizado foi possível produzir o seguinte diagrama UML simplificado, contendo as classes importantes no que diz respeito á este trabalho:

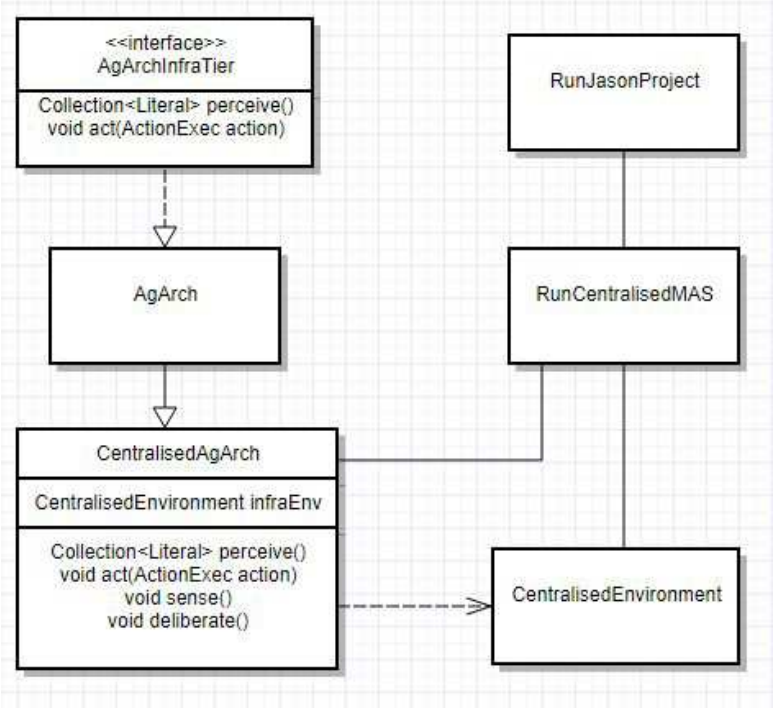


Figura 1. Diagrama de classes simplificado da arquitetura JASON

Dado este estudo inicial foi possível concluir que o JASON roda uma arquitetura onde o ambiente exerce um papel ativo, sendo o responsável por produzir as percepções do agente através do método getPercepts(String).

Ou seja, neste tipo de relacionamento o ambiente assume totalmente a responsabilidade de manter o agente atualizado com as informações que ele precisa, isto faz com que o ambiente precise atualizar os agentes com as modificações que ocorreram no ambiente - até quando os agentes não precisam de atualizações sobre o mesmo. Neste trabalho descreve-se este tipo de arquitetura como uma arquitetura de ambiente ativo. Este fluxo onde o ambiente alimenta o agente pode ser funcional e prático para alguns tipos de trabalhos com agentes, principalmente aqueles onde o ambiente fica dentro do próprio framework do agente, estes trabalhos em sua maioria são simulações simples. Porém quando amplia-se o contexto de atuação de agentes, como por exemplo agentes que atuam no mundo real (através de robôs, por exemplo) ou então em simulações onde o framework do ambiente roda em outra plataforma, fora do framework do agente isto torna-se um problema, pois conceitualmente não faz sentido "o mundo real" alimentar o agente com informações, e sim faz sentido o agente observar e angariar as informações que são de seu interesse. É exatamente neste ponto que contextualizam-se as idéias propostas por este trabalho.

A princípio esta questão tem uma implicação apenas conceitual, mas observando na prática nota-se que também existe uma implicação prática: em um cenário onde agentes atuariam no mundo real, não existe uma forma do mundo real alimentar o agente com informações, e seria necessário o desenvolvimento de um software separado que fosse responsável por observar o mundo real e alimentar o agente com as informações. O mesmo tipo de problema ocorreria em ambientes virtuais, cujo framework do ambiente é separado do framework do agente.

Observa-se que em casos onde tem-se um ambiente mais simples como em casos de interface gráfica muito simples (ex: um tabuleiro de jogo da velha) ou mesmo sem interface gráfica existe uma vantagem prática de ter este tipo de associação: a simplicidade na implementação. Já que o ambiente é simples e pode ser construído com facilidade no próprio framework dos agentes, é mais fácil construir as percepções do agente dentro do próprio ambiente e alimentá-los com elas.

Todavia outras implicações práticas negativas deste tipo de relacionamento são encontradas no código da arquitetura do agente e do ambiente:

- Falha na modelagem conceitual: Uma das atividades realizadas pelo ambiente consiste em processar a si mesmo de forma a construir as percepções de cada agente individualmente e alimentá-los com elas. Esta atividade claramente não é de sua responsabilidade, pois não é papel do ambiente alimentar o agente com informações e muito menos entender detalhes da construção das percepções dos agentes.
- Falha no encapsulamento: O encapsulamento na programação orientada a objetos consiste em ocultar detalhes do funcionamento interno de um determinado componente de forma que outros componentes que utilizam-o não precisem ter conhecimento do seu funcionamento para utilizá-lo. No caso da arquitetura de agentes temos a necessidade de expor detalhes da implementação da arquitetura de forma que o ambiente possa se comunicar com ela.

Visando sanar esta problemática este trabalho propõe o seguinte fluxo de relacionamento entre agente e ambiente:

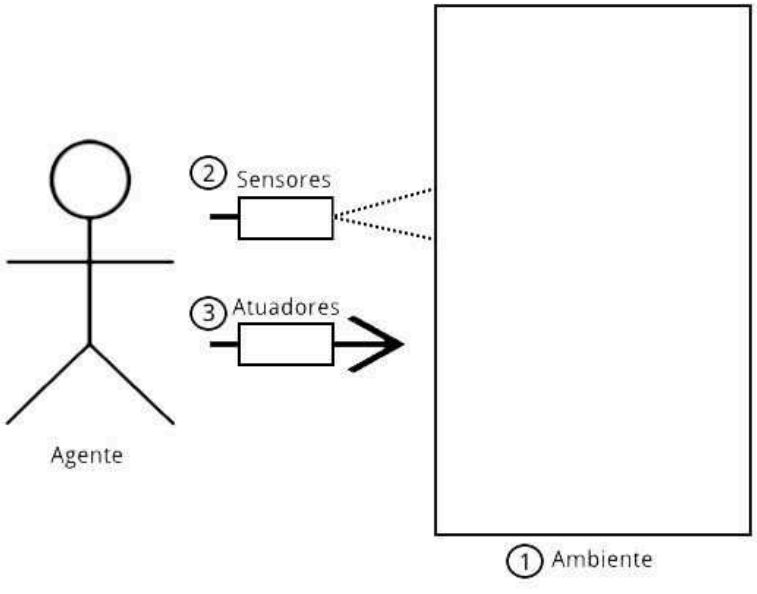


Figura 2. Relacionamento agente vs. ambiente passivo

Através da figura acima observa-se que o fluxo proposto é sempre na direção do ambiente, isto se dá pois o ambiente desempenha um papel totalmente passivo (caracterizando esta arquitetura como uma arquitetura de ambiente passivo), se atendo

totalmente apenas às suas funções nativas, deixando que o agente atue e observe-o da maneira e no tempo adequados. Desta maneira respeita-se o conceito de encapsulamento no código. Vendo a imagem de um ponto de vista mais técnico pode-se observar também a presença de dois novos componentes: os sensores (número 2) e atuadores (número 3). Estes componentes são os responsáveis por desacoplar do agente os comportamentos de observar e atuar sobre o ambiente, criando assim uma arquitetura mais flexível e dinâmica que, assim como o ambiente, respeita o conceito de encapsulamento.

O sensor do agente é responsável por implementar a maneira como ele observa o ambiente, implementando os detalhes da "comunicação" com o ambiente, e exercendo as operações necessárias sobre a observação realizada, de modo a interpretar os dados e retorná-los em forma de crenças para o agente. O atuador, da mesma maneira, precisa realizar a mesma comunicação com o ambiente e é responsável por implementar os detalhes de como o agente deve atuar sobre ele.

Sendo assim, se fez necessário uma modificação no fluxo de operação da arquitetura do agente, para torná-la uma arquitetura de ambiente passivo.

4.2. Modificação da arquitetura JASON

O seguinte passo, visando tornar o ambiente passivo, foi desacoplar o ambiente da arquitetura do agente (CentralisedAgArch) que possui como um de seus atributos um objeto da classe CentralisedEnvironment, este objeto foi removido no processo de desacoplamento, junto com todas as linhas de código que utilizavam-no. Uma vez removidas as dependências ambos os métodos act(ActionExec) e perceive() param de funcionar, já que estes são diretamente dependentes do ambiente. Então foi realizado um teste simples, fazendo com que o método perceive() retornasse uma lista vazia de percepções e que o método act(ActionExec) não realizasse nada. Feito isso foi executado um projeto simples de teste, com um agente sem desejos para garantir que o motor continuava rodando mesmo sem novas percepções. Como esperado o motor rodou sem apresentar erros.

Foi então dado início a implementação dos componentes atuador e sensor. Seguindo a idéia de que ambos os componentes não implementam um comportamento em si, apenas definem o padrão a ser seguido foi optado por definir interfaces com os nomes Actuator e Sensor. Com os respectivos métodos:

- Actuator: Representa o componente atuador. A classe que implementa a interface Actuator deve implementar um método com a assinatura act(ActionExec), sendo assim será responsável por implementar todo o processo de execução de uma ação;
- Sensor: Representa um sensor, e a classe que implementa-a deve implementar o método de assinatura perceive() e fica responsável por implementar o processo de captação das percepções do agente e retorná-las em forma de coleção;

Definido o modo como os componentes devem ser implementados o próximo passo foi decidir como esses componentes seriam injetados dentro da arquitetura do agente, representada pela classe CentralisedAgArch. Para isso foi necessário um novo estudo sobre a arquitetura do JASON buscando entender como é dado o processo de criação dos agentes a partir dos arquivos de definição .asl.

Foi iniciada uma busca objetivando localizar onde são criadas as instâncias da classe CentralisedAgArch, que representam a arquitetura de cada gente individualmente, para que

fosse possível parametriza-lá com a instância do sensor e do atuador do respectivo agente. Foi identificado que o método responsável por criar as instâncias é o createAgs() da classe RunCentralisedMAS. Sendo assim foi criado um novo método nesta mesma classe chamado loadDefinition(String), que é responsável por buscar na raiz do projeto JASON uma classe com o nome do agente no formato nomeDoAgente.java e esta classe deve implementar a interface AgenteDefinition. Ou seja, para cada agente criado no projeto deve-se criar uma classe .java com o mesmo nome do agente, e esta classe ficará responsável por instanciar o atuador e o sensor do agente. Uma vez carregada a definição do agente através deste método o próximo passo foi criar dois parâmetros no construtor da classe CentralisedAgArch para que esta possa receber o atuador e o sensor como parâmetros.

Tratando da classe CentralisedAgArch que agora recebe sensor e atuador como parâmetros, ambos os parâmetros foram alocados em atributos para que possam ser chamados quando necessários. A utilização do atuador se deu no método act(ActionExec) (que até o presente momento se encontrava vazio, devido ao desacoplamento do ambiente) de forma que agora o método tem apenas de chamar o método act(ActionExec) do atuador. A sua implementação se dá de forma simples já que agora a responsabilidade de implementar como a ação é executada cabe apenas ao atuador.

Desta forma a nova arquitetura pode ser representada pelo diagrama:

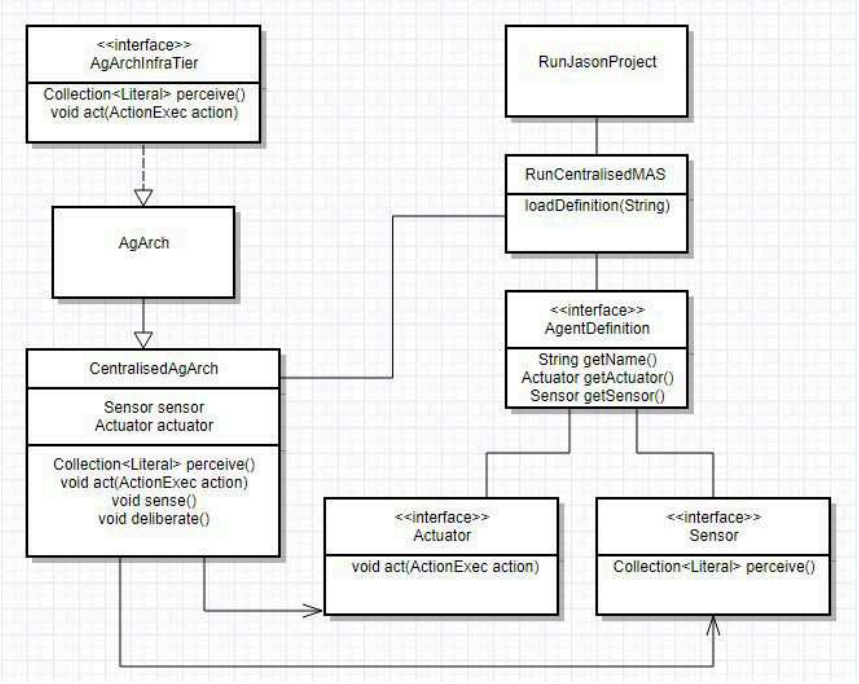


Figura 3. Diagrama de classes simplificado da nova arquitetura

Uma vez implementada a nova arquitetura foi dado início a uma extensiva bateria de testes preliminares utilizando diferentes cenários de testes, objetivando a validação da arquitetura. Os erros encontrados foram corrigidos de forma que foi possível concluir que a arquitetura e o motor BDI funcionavam corretamente em tais cenários.

5. Estudo de caso no projeto Awareness

Conforme descrito anteriormente este trabalho tem como sua motivação inicial o projeto Awareness e também tem como um de seus objetivos específicos a realização de um estudo de caso da arquitetura desenvolvida nele. O projeto Awaraness tem como objetivo fazer um estudo acerca do foco de pessoas enquanto estas fazem uso de dispositivos móveis em situações como o trânsito de automóveis. Para realizar este estudo faz-se uso de uma simulação de realidade virtual 3D desenvolvida na plataforma Unity, que é uma arquitetura para desenvolvimento de cenários virtuais. Esta simulação consiste de um cenário de uma cidade urbana, onde tem-se uma via de automóveis, uma faixa de pedestres, um semáforo para os pedestres e carros. A visão do humano fica localizado próximo a faixa de pedestres e sua função dentro da simulação é controlar o semáforo de forma que os pedestres (representados por gatos) não colidam com os carros, alterando o sinal para vermelho quando não é seguro atravessar e alterando para verde quando é seguro.

Figura 4. Simulação imersiva de realidade virtual do projeto Awareness

De forma a simular distrações reais, a pessoa que está participando da simulação opera um smartphone e distrações programadas são enviadas para o aparelho de forma a poder detectar o impacto que estas distrações tem no foco da pessoa, que é responsável em paralelo por controlar o semáforo da simulação. Observe que, já que o objetivo da simulação é fazer uma análise acerca do foco nos seres humanos para que se obtenha resultados com fidelidade, é necessário uma simulação com um suficiente nível de representatividade, sendo assim os componentes da simulação (motoristas, pedestres, etc) devem possuir este nível, e para este tipo de comportamento computacional considera-se adequada a utilização de agentes. Porém é neste ponto que encontra-se a problemática que este trabalho aborda: as arquiteturas atuais para o desenvolvimento de agentes funcionam com ambientes mais simples, onde o ambiente encontra-se e controlado pelo próprio framework do agente, o que não é o caso do Awareness. Então, para isto, utiliza-se a arquitetura que este trabalho desenvolveu, de forma que os agentes são desenvolvidos em Java na arquitetura JASON e são integrados à simulação em Unity através da estrutura de atuadores e sensores que utiliza uma conexão socket. Sendo assim optou-se por realizar a implementação de um agente que representa o pedestre (gato) na simulação, o comportamento do pedestre consiste basicamente

em parar de caminhar quando encontra o semáforo fechado e voltar a caminhar quando o mesmo está aberto.

O desenvolvimento do teste se deu em duas partes, sendo a primeira delas o desenvolvimento no que diz respeito aos agentes, em Java. Para isso foi criado um projeto JASON chamado awareness_lemming. Neste projeto o primeiro passo foi criar a definição do agente chamado lemming no arquivo lemming.asl. Para realizar a definição utilizou-se a lógica de eventos: eventos são disparados ao serem adicionadas novas crenças ao agente: quando a crença redLight é adicionada ao agente, o mesmo adiciona na lista de intenções a intenção stopWalking, e quando a crença greenLight é adicionado adiciona a intenção startWalking.

O proximo passo foi a criação da classe lemming.java, que implementa a interface AgentDefinition, de forma a informar o atuador e o sensor do agente, representados pelas classes LemmingUnityActuator e LemmingUnitySensor, respectivamente. Ambos se comunicam com o Unity através de uma comunicação Socket implementada em um dos testes preliminares, que é basicamente um protocolo simples de comunicação que permite que dois programas diferentes (independente de sua linguagem de implementação) comuniquem-se. O atuador pode enviar duas mensagens diferentes de acordo com a ação a ser executada, que é determinada pelas intenções do agente:

- No caso de uma intenção startWalking o atuador envia uma mensagem com o conteúdo “1” de forma a comunicar para o ambiente que o gato deve começar a andar - é necessário apenas enviar um sinal para que ele comece a andar, já que a implementação do ato de “andar” de fato (movimentar-se, escolher a direção) é implementada pelo próprio projeto Unity.
- No caso de uma intenção stopWalking o atuador envia a mensagem com o conteúdo “2” que comunica que o gato deve parar de andar;

Já o sensor sempre envia a mensagem com o conteúdo “0” e aguarda a resposta do ambiente. O conteúdo “0” indica ao Unity que quer-se saber se o semáforo está com a cor vermelha ligada. Desta forma o sensor verifica a cor do semáforo e traduz isso para uma crença: se a resposta for positiva (luz vermelha ativa) ele retorna a crença redLight e caso seja negativa (luz verde ativa) retorna a crença greenLight.

Desenvolvido o projeto JASON o próximo passo foi a implementação do lado Unity, um servidor socket que deve responder as mensagens que possuem quatro possíveis conteúdos:

- Requisição com conteúdo “0”: No caso desta requisição que busca saber se o semáforo está com a cor vermelha ativa, foi utilizado o componente PuffinCrossing que representa o semáforo, nele existe o método isRed(). Desta forma basta realizar a chamada deste método e retornar o resultado.
- Requisição com conteúdo “1”: Já nesta requisição deve-se fazer com que o gato comece a andar. A classe MoveToWayPointsLemming possui o método move(), que é responsável por implementar de fato a movimentação do gato. Este método foi modificado de forma a adicionar uma verificação em uma variável booleana isMoving, de forma que se esta variável seja setada para falso o gato não realiza a movimentação, permanecendo parado. Bastou então adicionar nesta classe métodos que permitem alterar para verdadeiro ou falso a variável booleana. Sendo assim, quando o servidor socket recebe esta

requisição, ele tem apenas de fazer a chamada do método que seta a variável para verdadeiro.

- Requisição com conteúdo “2”: Esta requisição deve fazer com o que o gato pare de andar. Utiliza exatamente a mesma estrutura da requisição anterior, porém realiza a chamada do método que seta a variável para falso.
- Requisição com conteúdo “3”: Esta requisição não é utilizada pelo agente, e foi inserida para tornar possível a realização do teste: ela é responsável por alterar a cor atual do semáforo, alterando para verde caso esteja vermelha e vice-versa. Para realizar isto foi necessário apenas chamar o método isRed() do componente PuffinCrossing verificando se o sinal está com a luz vermelha acesa, e caso esteja chamada o método changeToGreen() e caso contrário chamada o método changeToRed().

Para enviar uma mensagem com o conteúdo “3” foi desenvolvido uma classe java chamada UnitySemaforoController, que possui apenas um método main responsável apenas por enviar a mensagem através da mesma conexão socket utilizada pelo atuador e pelo sensor. Sendo assim esta classe pode ser executada a qualquer momento.

Feito o desenvolvimento de ambos os lados do teste, o mesmo estava pronto para ser executado. Para realizar o teste primeiro foi iniciada a simulação, e então foi iniciado o motor BDI do projeto JASON. Inicialmente o gato começou a caminhar sem problemas, e caminhava livremente pela faixa de pedestres, independente da presença de carros - já que o semáforo é iniciado sempre com a luz verde. No momento em que foi executada a classe UnitySemaforoController, mudando a luz para vermelho, o seguinte ocorreu:

- A crença redLight foi adicionada a lista de crenças do agente;
- O agente enviou uma requisição ao Unity para que o gato pare de andar;
- O gato parou de andar.

Ao executar a classe UnitySemaforoController novamente mudando a luz para verde, o mesmo processo descrito anteriormente ocorreu, porém com a crença greenLight e uma requisição para que o gato volte a andar e como esperado o gato voltou a andar. Sendo assim conclui-se que o teste foi um sucesso e que o agente comportou-se exatamente como esperado. Através deste teste foi possível identificar que o motor BDI continua a funcionar corretamente mesmo em um contexto onde o framework do ambiente está separado do framework do agente, concluindo assim o último objetivo específico deste trabalho que consiste em realizar um estudo de caso acerca da implementação realizada, modelando agentes para representar os pedestres na simulação imersiva do projeto Awareness.

6. Conclusão

Este trabalho inicialmente destacou no capítulo 1 uma problemática presente nas arquiteturas de agentes atualmente disponíveis, buscou embasar-se na literatura sobre agentes e suas possíveis implementações no capítulo possibilitando selecionar qual seria a melhor teoria (capítulo 3) e a melhor arquitetura de agentes (capítulo 4) a serem utilizadas para o desenvolvimento da arquitetura a qual este trabalho se propõe a realizar. No capítulo 5 foi implementada a arquitetura, realizado uma série de testes e correções sobre a mesma e ao final foi realizado um estudo de caso.

Considerando-se os testes realizados na arquitetura pode-se concluir que a arquitetura implementada cumpre o objetivo a qual este trabalho se propõe realizar, que é tornar factível o desenvolvimento de agentes cujo ambiente está fora do framework do próprio agente. Isto é

obtido através dos componentes atuador e sensor, pois ocorre uma correta divisão de responsabilidade, fazendo com que o agente seja responsável apenas por implementar os componentes do BDI, com que o ambiente seja responsável apenas por implementar o que diz respeito a si mesmo, enquanto o atuador e sensor ficam responsáveis por implementar a forma de interação com o ambiente e a análise dos dados recebidos do ambiente de forma a criar as percepções.

Por consequência da correta divisão de responsabilidade tem-se como resultado uma arquitetura que dá suporte ao cenário do projeto Awareness, e também aos mais diversos cenários, expandindo de forma considerável a aplicabilidade do JASON. Além disso a arquitetura também promove boas práticas de desenvolvimento de software no que diz respeito à programação orientada a objetos, respeitando o conceito de encapsulamento, aumentando a manutenibilidade e testabilidade dos projetos desenvolvidos nela. Conclui-se também que já que a forma de comunicação com o ambiente, encontra-se totalmente isolada no atuador e sensor, é possível mudar completamente o ambiente onde um agente atua, modificando apenas o atuador e sensor sem ter de modificar nada na implementação do agente.

Uma limitação identificada na arquitetura é no caso de uma possível demora na resposta de uma requisição ao ambiente, quando se está por exemplo buscando informações para construir as crenças do agente. Neste caso já que o motor BDI precisa das crenças do agente para executar o ciclo de pensamento do agente, a chamada deste método é síncrona e o ciclo de pensamento terá de esperar até que o ambiente responda a requisição.

Notou-se também que apesar deste trabalho propor uma arquitetura para atuação de agentes em ambientes virtuais, a arquitetura desenvolvida é flexível ao ponto de poder ser aplicada também para a atuação de agentes em ambientes reais, em casos como por exemplo onde temos um agente virtual, controlando um objeto real (como um robô).

6. Trabalhos futuros

Uma primeira sugestão de trabalho futuro é abordar a problemática descrita na conclusão: o que fazer no caso de demora na resposta do ambiente na busca pelas percepções do agente? Em casos onde o sensor do agente tem de buscar algum dado no ambiente, pode existir alguma demora no tempo de resposta, e seria interessante modificar o motor BDI de forma que o agente pudesse vir a continuar o seu ciclo de raciocínio enquanto aguarda a resposta.

Uma segunda sugestão é para casos onde o agente tem de criar suas percepções através da observação do ambiente como um todo, ou seja, em vez de perguntar dados específicos ao ambiente como realizado no estudo de caso, o agente iria apenas pedir o que está em seu campo de visão e realizar uma interpretação dos dados recebidos de forma a construir as suas percepções. Isto permite que seja desenvolvido agentes de comportamento enriquecido, já que as percepções do agente sobre o ambiente são muito mais abrangentes. Desta forma seria de grande valor ter um novo componente acoplado ao sensor do agente, de forma que o sensor seja apenas responsável por implementar a comunicação com o ambiente, e passar os dados “brutos” recebidos para tal componente e que ele realize a tradução destes dados para crenças do agente.

Referências

WOOLDRIDGE, Michael. Intelligent Agents: The Key Concepts. Department of Computer Science. University of Liverpool. 2002.

BONSON, Jéssica. Aplicação de agentes & artefatos para o desenvolvimento de uma ferramenta de autoria de objetos de aprendizagem. 2012.

WOOLDRIDGE, Michael; JENNINGS, Nicholas. Agent Theories, Architectures, and Languages: A Survey. 1995.

WOOLDRIDGE, Michael. Intelligent Agents: The Key Concepts. Department of Computer Science. University of Liverpool. 2002.